The module must also be able to recognize and generate addresses associated with the devices it controls. Each I/O module has a unique address or, if it controls more than one external device, a unique set of addresses. Finally, the I/O module contains logic specific to the interface with each device that it controls.

An I/O module functions to allow the processor to view a wide range of devices in a simple-minded way. There is a spectrum of capabilities that may be provided. The I/O module may hide the details of timing, formats, and the electromechanics of an external device so that the processor can function in terms of simple read and write commands, and possibly open and close file commands. In its simplest form, the I/O module may still leave much of the work of controlling a device (e.g., rewind a tape) visible to the processor.

An I/O module that takes on most of the detailed processing burden, presenting a high-level interface to the processor, is usually referred to as an *I/O channel* or *I/O processor.* An I/O module that is quite primitive and requires detailed control is usually referred to as an *I/O controller* or *device controller.* I/O controllers are commonly seen on microcomputers, whereas I/O channels are used on mainframes.

In what follows, we will use the generic term *I/O module* when no confusion results and will use more specific terms where necessary.

7.3 PROGRAMMED I/O

Three techniques are possible for I/O operations. With *programmed I/O,* data are exchanged between the processor and the I/O module. The processor executes a program that gives it direct control of the I/O operation, including sensing device status, sending a read or write command, and transferring the data. When the processor issues a command to the I/O module, it must wait until the I/O operation is complete. If the processor is faster than the I/O module, this is wasteful of processor time. With *interrupt-driven I/O,* the processor issues an I/O command, continues to execute other instructions, and is interrupted by the I/O module when the latter has completed its work. With both programmed and interrupt I/O, the processor is responsible for extracting data from main memory for output and storing data in main memory for input. The alternative is known as *direct memory access* (DMA). In this mode, the I/O module and main memory exchange data directly, without processor involvement.

Table 7.3 indicates the relationship among these three techniques. In this section, we explore programmed I/O. Interrupt I/O and DMA are explored in the following two sections, respectively.

Table 7.3 I/O Techniques

	No Interrupts	Use of Interrupts
I/O-to-memory transfer through processor	Programmed I/O	Interrupt-driven I/O
Direct I/O-to-memory transfer		Direct memory access (DMA)

Overview of Programmed I/O

When the processor is executing a program and encounters an instruction relating to I/O, it executes that instruction by issuing a command to the appropriate I/O module. With programmed I/O, the I/O module will perform the requested action and then set the appropriate bits in the I/O status register (Figure 7.3). The I/O module takes no further action to alert the processor. In particular, it does not interrupt the processor. Thus, it is the responsibility of the processor periodically to check the status of the I/O module until it finds that the operation is complete.

To explain the programmed I/O technique, we view it first from the point of view of the I/O commands issued by the processor to the I/O module, and then from the point of view of the I/O instructions executed by the processor.

I/O Commands

To execute an I/O-related instruction, the processor issues an address, specifying the particular I/O module and external device, and an I/O command. There are four types of I/O commands that an I/O module may receive when it is addressed by a processor:

- **Control:** Used to activate a peripheral and tell it what to do. For example, a magnetic-tape unit may be instructed to rewind or to move forward one record. These commands are tailored to the particular type of peripheral device.
- **Test:** Used to test various status conditions associated with an I/O module and its peripherals. The processor will want to know that the peripheral of interest is powered on and available for use. It will also want to know if the most recent I/O operation is completed and if any errors occurred.
- **Read:** Causes the I/O module to obtain an item of data from the peripheral and place it in an internal buffer (depicted as a data register in Figure 7.3). The processor can then obtain the data item by requesting that the I/O module place it on the data bus.
- **Write:** Causes the I/O module to take an item of data (byte or word) from the data bus and subsequently transmit that data item to the peripheral.

Figure 7.4a gives an example of the use of programmed I/O to read in a block of data from a peripheral device (e.g., a record from tape) into memory. Data are read in one word (e.g., 16 bits) at a time. For each word that is read in, the processor must remain in a status-checking cycle until it determines that the word is available in the I/O module's data register. This flowchart highlights the main disadvantage of this technique: It is a time-consuming process that keeps the processor busy needlessly.

I/O Instructions

With programmed I/O, there is a close correspondence between the I/O-related instructions that the processor fetches from memory and the I/O commands that the processor issues to an I/O module to execute the instructions. That is, the instructions are easily mapped into I/O commands, and there is often a simple one-to-one relationship. The form of the instruction depends on the way in which external devices are addressed.

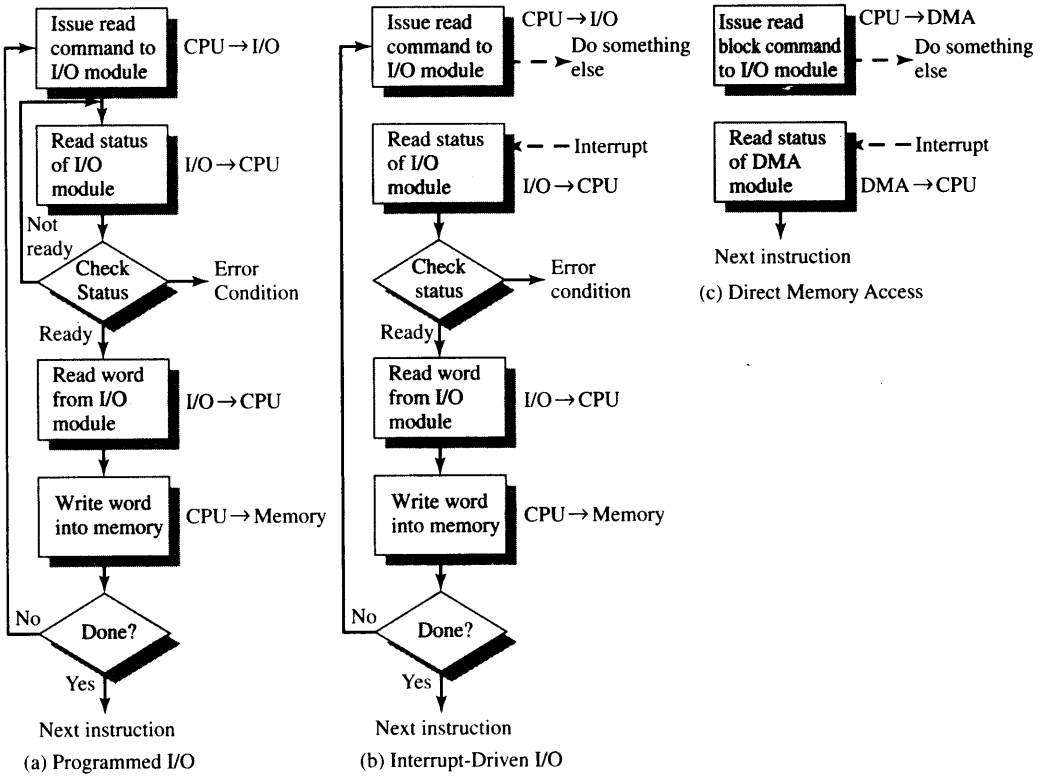


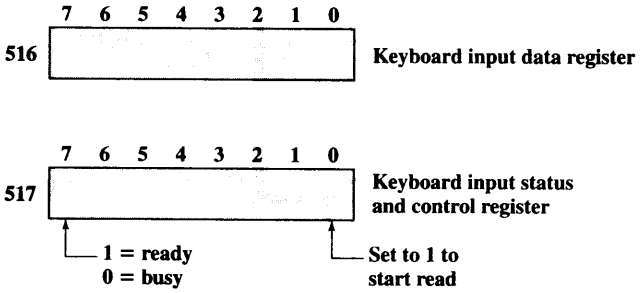
Figure 7.4 Three Techniques for Input of a Block of Data

Typically, there will be many I/O devices connected through I/O modules to the system. Each device is given a unique identifier or address. When the processor issues an I/O command, the command contains the address of the desired device. Thus, each I/O module must interpret the address lines to determine if the command is for itself.

When the processor, main memory, and I/O share a common bus, two modes of addressing are possible: memory mapped and isolated. With **memory-mapped I/O**, there is a single address space for memory locations and I/O devices. The processor treats the status and data registers of I/O modules as memory locations and uses the same machine instructions to access both memory and I/O devices. So, for example, with 10 address lines, a combined total of $2^{10} = 1024$ memory locations and I/O addresses can be supported, in any combination.

With memory-mapped I/O, a single read line and a single write line are needed on the bus. Alternatively, the bus may be equipped with memory read and write plus input and output command lines. Now, the command line specifies whether the address refers to a memory location or an I/O device. The full range of addresses may be available for both. Again, with 10 address lines, the system may now support both 1024 memory locations and 1024 I/O addresses. Because the address space for I/O is isolated from that for memory, this is referred to as **isolated I/O**.

Figure 7.5 contrasts these two programmed I/O techniques. Figure 7.5a shows how the interface for a simple input device such as a terminal keyboard might appear



ADDRESS	INSTRUCTION	OPERAND	COMMENT
200	Load AC	"1"	Load accumulator
	Store AC	517	Initiate keyboard read
202	Load AC	517	Get status byte
	Branch if Sign = 0	202	Loop until ready
	Load AC	516	Load data byte

(a) Memory-mapped I/O

ADDRESS	INSTRUCTION	OPERAND	COMMENT
200	Load I/O	5	Initiate keyboard read
201	Test I/O	5	Check for completion
	Branch Not Ready	201	Loop until complete
	In	5	Load data byte

(b) Isolated I/O

Figure 7.5 Memory-Mapped and Isolated I/O

to a programmer using memory-mapped I/O. Assume a 10-bit address, with a 512-bit memory (locations 0–511) and up to 512 I/O addresses (locations 512–1023). Two addresses are dedicated to keyboard input from a particular terminal. Address 516 refers to the data register and address 517 refers to the status register, which also functions as a control register for receiving processor commands. The program shown will read 1 byte of data from the keyboard into an accumulator register in the processor. Note that the processor loops until the data byte is available.

With isolated I/O (Figure 7.5b), the I/O ports are accessible only by special I/O commands, which activate the I/O command lines on the bus.

For most types of processors, there is a relatively large set of different instructions for referencing memory. If isolated I/O is used, there are only a few I/O instructions. Thus, an advantage of memory-mapped I/O is that this large repertoire of instructions can be used, allowing more efficient programming. A disadvantage is that valuable memory address space is used up. Both memory-mapped and isolated I/O are in common use.

7.4 INTERRUPT-DRIVEN I/O

The problem with programmed I/O is that the processor has to wait a long time for the I/O module of concern to be ready for either reception or transmission of data.

The processor, while waiting, must repeatedly interrogate the status of the I/O module. As a result, the level of the performance of the entire system is severely degraded.

An alternative is for the processor to issue an I/O command to a module and then go on to do some other useful work. The I/O module will then interrupt the processor to request service when it is ready to exchange data with the processor. The processor then executes the data transfer, as before, and then resumes its former processing.

Let us consider how this works, first from the point of view of the I/O module. For input, the I/O module receives a READ command from the processor. The I/O module then proceeds to read data in from an associated peripheral. Once the data are in the module's data register, the module signals an interrupt to the processor over a control line. The module then waits until its data are requested by the processor. When the request is made, the module places its data on the data bus and is then ready for another I/O operation.

From the processor's point of view, the action for input is as follows. The processor issues a READ command. It then goes off and does something else (e.g., the processor may be working on several different programs at the same time). At the end of each instruction cycle, the processor checks for interrupts (Figure 3.9). When the interrupt from the I/O module occurs, the processor saves the context (e.g., program counter and processor registers) of the current program and processes the interrupt. In this case, the processor reads the word of data from the I/O module and stores it in memory. It then restores the context of the program it was working on (or some other program) and resumes execution.

Figure 7.4b shows the use of interrupt I/O for reading in a block of data. Compare this with Figure 7.4a. Interrupt I/O is more efficient than programmed I/O because it eliminates needless waiting. However, interrupt I/O still consumes a lot of processor time, because every word of data that goes from memory to I/O module or from I/O module to memory must pass through the processor.

Interrupt Processing

Let us consider the role of the processor in interrupt-driven I/O in more detail. The occurrence of an interrupt triggers a number of events, both in the processor hardware and in software. Figure 7.6 shows a typical sequence. When an I/O device completes an I/O operation, the following sequence of hardware events occurs:

1. The device issues an interrupt signal to the processor.
2. The processor finishes execution of the current instruction before responding to the interrupt, as indicated in Figure 3.9.
3. The processor tests for an interrupt, determines that there is one, and sends an acknowledgment signal to the device that issued the interrupt. The acknowledgment allows the device to remove its interrupt signal.
4. The processor now needs to prepare to transfer control to the interrupt routine. To begin, it needs to save information needed to resume the current program at the point of interrupt. The minimum information required is (a) the status of the processor, which is contained in a register called the program status word (PSW),

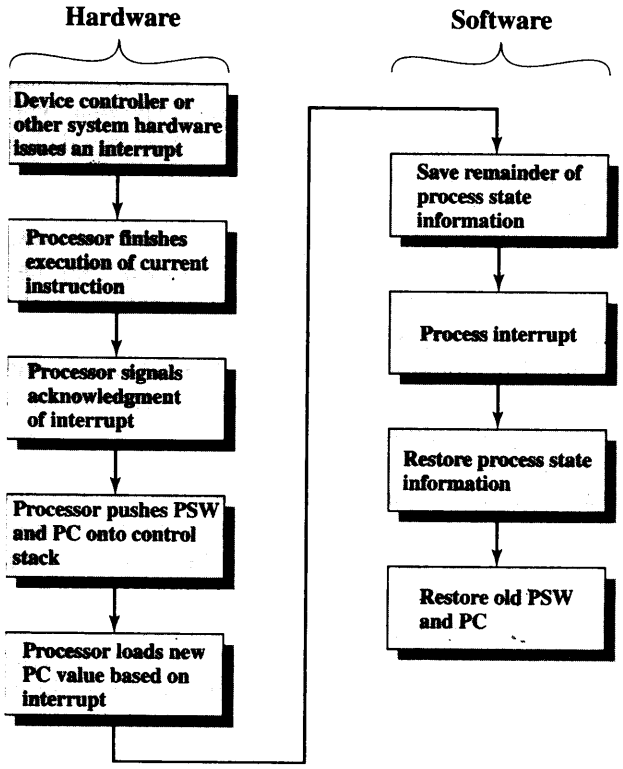


Figure 7.6 Simple Interrupt Processing

and (b) the location of the next instruction to be executed, which is contained in the program counter. These can be pushed onto the system control stack.[3]

5. The processor now loads the program counter with the entry location of the interrupt-handling program that will respond to this interrupt. Depending on the computer architecture and operating system design, there may be a single program; one program for each type of interrupt; or one program for each device and each type of interrupt. If there is more than one interrupt-handling routine, the processor must determine which one to invoke. This information may have been included in the original interrupt signal, or the processor may have to issue a request to the device that issued the interrupt to get a response that contains the needed information.

Once the program counter has been loaded, the processor proceeds to the next instruction cycle, which begins with an instruction fetch. Because the instruction fetch is determined by the contents of the program counter, the result is that control is transferred to the interrupt-handler program. The execution of this program results in the following operations:

[3]See Appendix 10A for a discussion of stack operation.

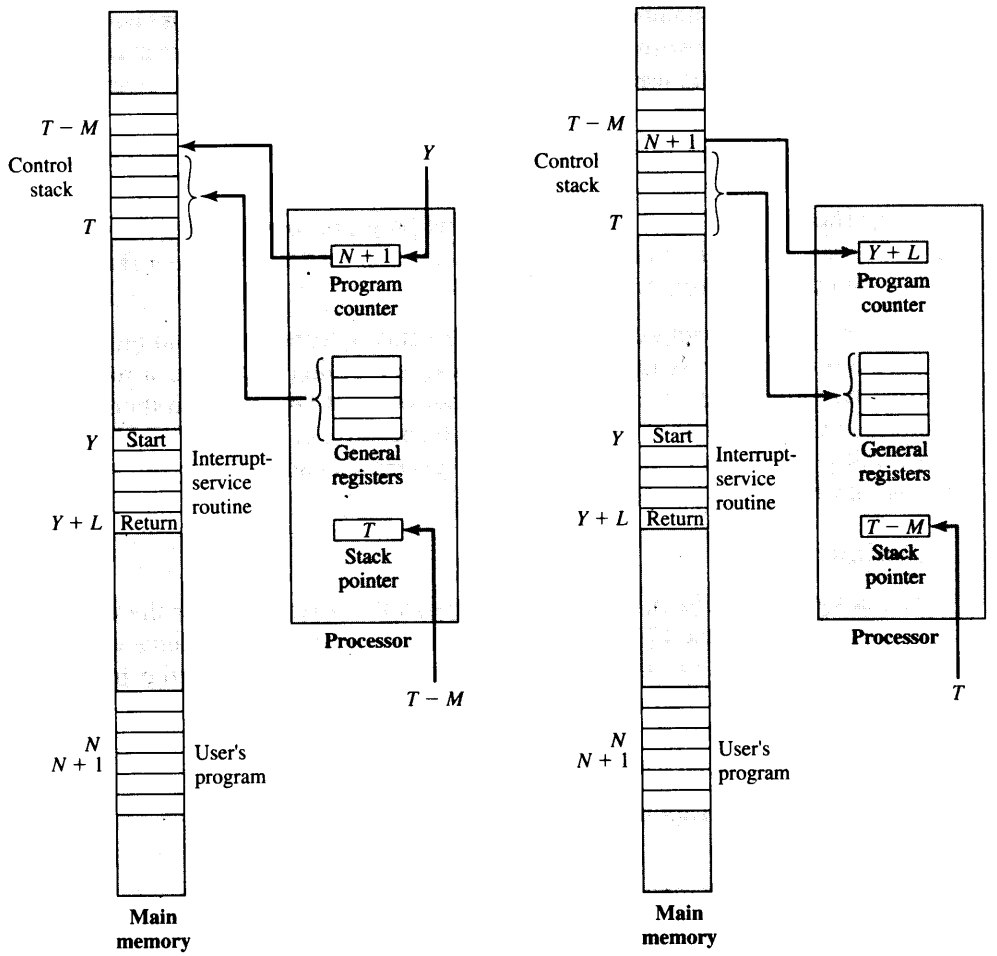


Figure 7.7 Changes in Memory and Registers for an Interrupt

6. At this point, the program counter and PSW relating to the interrupted program have been saved on the system stack. However, there is other information that is considered part of the "state" of the executing program. In particular, the contents of the processor registers need to be saved, because these registers may be used by the interrupt handler. So, all of these values, plus any other state information, need to be saved. Typically, the interrupt handler will begin by saving the contents of all registers on the stack. Figure 7.7a shows a simple example. In this case, a user program is interrupted after the instruction at location N. The contents of all of the registers plus the address of the next instruction ($N + 1$) are pushed onto the stack. The stack pointer is updated to point to the new top of stack, and the program counter is updated to point to the beginning of the interrupt service routine.

7. The interrupt handler next processes the interrupt. This includes an examination of status information relating to the I/O operation or other event that caused an interrupt. It may also involve sending additional commands or acknowledgments to the I/O device.
8. When interrupt processing is complete, the saved register values are retrieved from the stack and restored to the registers (e.g., see Figure 7.7b).
9. The final act is to restore the PSW and program counter values from the stack. As a result, the next instruction to be executed will be from the previously interrupted program.

Note that it is important to save all the state information about the interrupted program for later resumption. This is because the interrupt is not a routine called from the program. Rather, the interrupt can occur at any time and therefore at any point in the execution of a user program. Its occurrence is unpredictable. Indeed, as we will see in the next chapter, the two programs may not have anything in common and may belong to two different users.

Design Issues

Two design issues arise in implementing interrupt I/O. First, because there will almost invariably be multiple I/O modules, how does the processor determine which device issued the interrupt? And second, if multiple interrupts have occurred, how does the processor decide which one to process?

Let us consider device identification first. Four general categories of techniques are in common use:

- Multiple interrupt lines
- Software poll
- Daisy chain (hardware poll, vectored)
- Bus arbitration (vectored)

The most straightforward approach to the problem is to provide **multiple interrupt lines** between the processor and the I/O modules. However, it is impractical to dedicate more than a few bus lines or processor pins to interrupt lines. Consequently, even if multiple lines are used, it is likely that each line will have multiple I/O modules attached to it. Thus, one of the other three techniques must be used on each line.

One alternative is the **software poll**. When the processor detects an interrupt, it branches to an interrupt-service routine whose job it is to poll each I/O module to determine which module caused the interrupt. The poll could be in the form of a separate command line (e.g., TESTI/O). In this case, the processor raises TESTI/O and places the address of a particular I/O module on the address lines. The I/O module responds positively if it set the interrupt. Alternatively, each I/O module could contain an addressable status register. The processor then reads the status register of each I/O module to identify the interrupting module. Once the correct module is identified, the processor branches to a device-service routine specific to that device.

The disadvantage of the software poll is that it is time consuming. A more efficient technique is to use a **daisy chain**, which provides, in effect, a hardware poll. An example of a daisy-chain configuration is shown in Figure 3.26. For interrupts,

all I/O modules share a common interrupt request line. The interrupt acknowledge line is daisy chained through the modules. When the processor senses an interrupt, it sends out an interrupt acknowledge. This signal propagates through a series of I/O modules until it gets to a requesting module. The requesting module typically responds by placing a word on the data lines. This word is referred to as a *vector* and is either the address of the I/O module or some other unique identifier. In either case, the processor uses the vector as a pointer to the appropriate device-service routine. This avoids the need to execute a general interrupt-service routine first. This technique is called a *vectored interrupt.*

There is another technique that makes use of vectored interrupts, and that is **bus arbitration.** With bus arbitration, an I/O module must first gain control of the bus before it can raise the interrupt request line. Thus, only one module can raise the line at a time. When the processor detects the interrupt, it responds on the interrupt acknowledge line. The requesting module then places its vector on the data lines.

The aforementioned techniques serve to identify the requesting I/O module. They also provide a way of assigning priorities when more than one device is requesting interrupt service. With multiple lines, the processor just picks the interrupt line with the highest priority. With software polling, the order in which modules are polled determines their priority. Similarly, the order of modules on a daisy chain determines their priority. Finally, bus arbitration can employ a priority scheme, as discussed in Section 3.4.

We now turn to two examples of interrupt structures.

Intel 82C59A Interrupt Controller

The Intel 80386 provides a single Interrupt Request (INTR) and a single Interrupt Acknowledge (INTA) line. To allow the 80386 to handle a variety of devices and priority structures, it is usually configured with an external interrupt arbiter, the 82C59A. External devices are connected to the 82C59A, which in turn connects to the 80386.

Figure 7.8 shows the use of the 82C59A to connect multiple I/O modules for the 80386. A single 82C59A can handle up to 8 modules. If control for more than 8 modules is required, a cascade arrangement can be used to handle up to 64 modules.

The 82C59A's sole responsibility is the management of interrupts. It accepts interrupt requests from attached modules, determines which interrupt has the highest priority, and then signals the processor by raising the INTR line. The processor acknowledges via the INTA line. This prompts the 82C59A to place the appropriate vector information on the data bus. The processor can then proceed to process the interrupt and to communicate directly with the I/O module to read or write data.

The 82C59A is programmable. The 80386 determines the priority scheme to be used by setting a control word in the 82C59A. The following interrupt modes are possible:

- **Fully nested:** The interrupt requests are ordered in priority from 0 (IR0) through 7 (IR7).
- **Rotating:** In some applications a number of interrupting devices are of equal priority. In this mode a device, after being serviced, receives the lowest priority in the group.
- **Special mask:** This allows the processor to inhibit interrupts from certain devices.

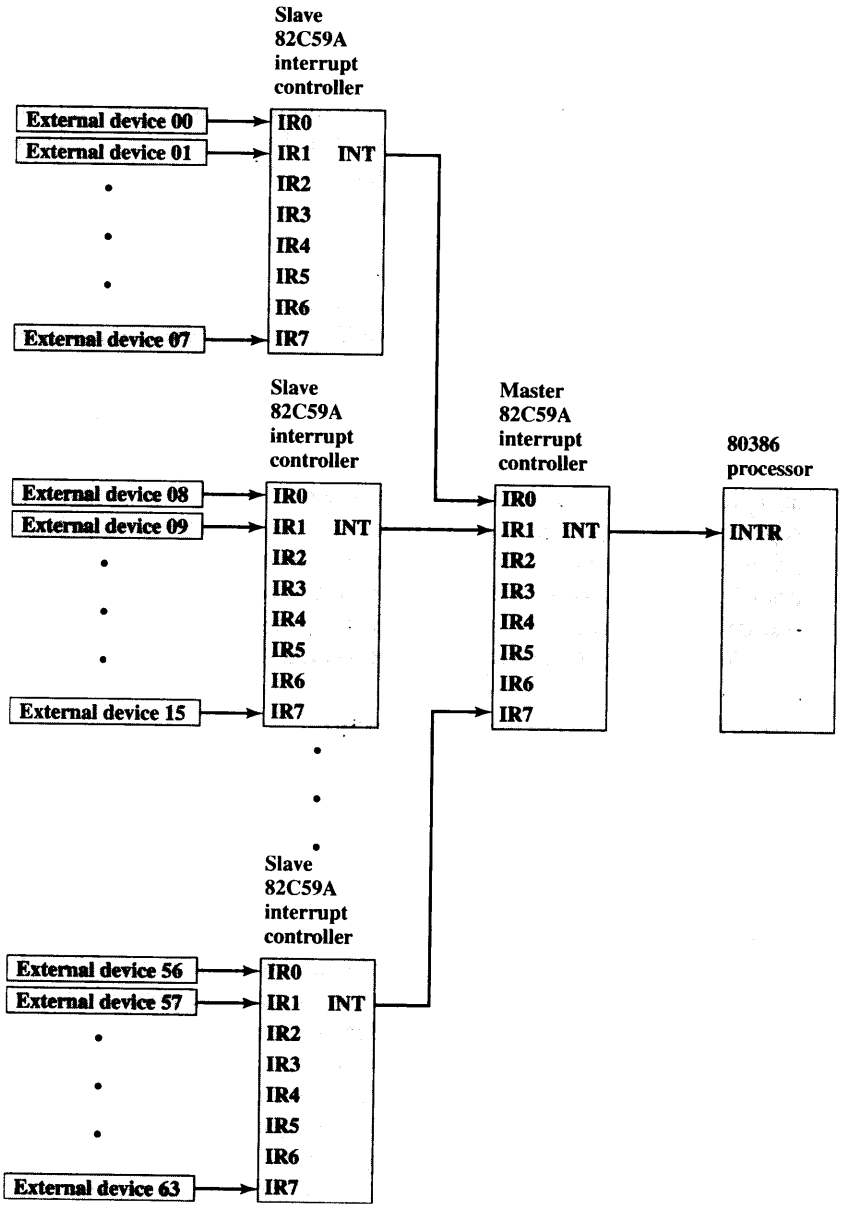
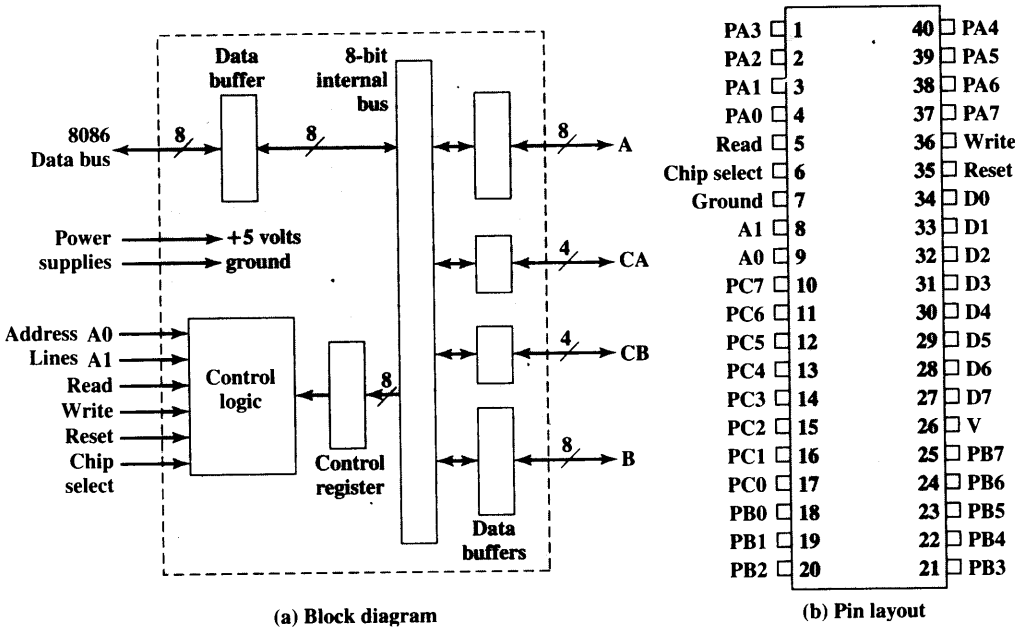


Figure 7.8 Use of the 82C59A Interrupt Controller

The Intel 82C55A Programmable Peripheral Interface

As an example of an I/O module used for programmed I/O and interrupt-driven I/O, we consider the Intel 82C55A Programmable Peripheral Interface. The 82C55A is a single-chip, general-purpose I/O module designed for use with the Intel 80386 processor. Figure 7.9 shows a general block diagram plus the pin assignment for the 40-pin package in which it is housed.



(a) Block diagram

(b) Pin layout

Figure 7.9 The Intel 82C55A Programmable Peripheral Interface

The right side of the block diagram is the external interface of the 82C55A. The 24 I/O lines are programmable by the 80386 by means of the control register. The 80386 can set the value of the control register to specify a variety of operating modes and configurations. The 24 lines are divided into three 8-bit groups (A, B, C). Each group can function as an 8-bit I/O port. In addition, group C is subdivided into 4-bit groups (C_A and C_B), which may be used in conjunction with the A and B I/O ports. Configured in this manner, they carry control and status signals.

The left side of the block diagram is the internal interface to the 80386 bus. It includes an 8-bit bidirectional data bus (D0 through D7), used to transfer data to and from the I/O ports and to transfer control information to the control register. The two address lines specify one of the three I/O ports or the control register. A transfer takes place when the CHIP SELECT line is enabled together with either the READ or WRITE line. The RESET line is used to initialize the module.

The control register is loaded by the processor to control the mode of operation and to define signals, if any. In Mode 0 operation, the three groups of eight external lines function as three 8-bit I/O ports. Each port can be designated as input or output. Otherwise, groups A and B function as I/O ports, and the lines of group C serve as control lines for A and B. The control signals serve two principal purposes: "handshaking" and interrupt request. Handshaking is a simple timing mechanism. One control line is used by the sender as a DATA READY line, to indicate when the data are present on the I/O data lines. Another line is used by the receiver as an ACKNOWLEDGE, indicating that the data have been read and the data lines may

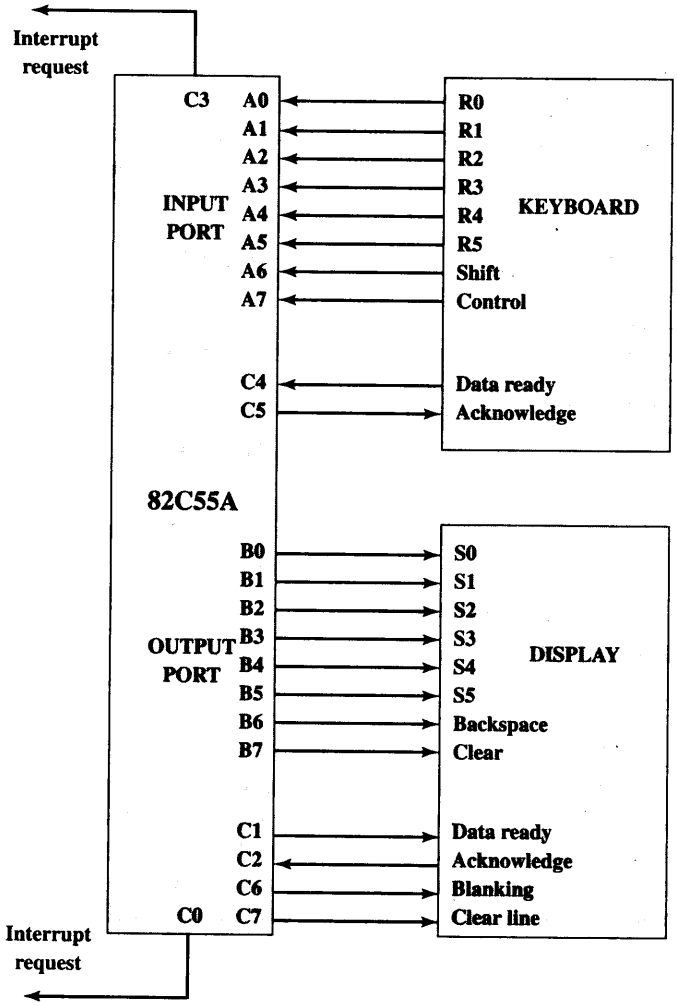


Figure 7.10 Keyboard/Display Interface to 82C55A

be cleared. Another line may be designated as an INTERRUPT REQUEST line and tied back to the system bus.

Because the 82C55A is programmable via the control register, it can be used to control a variety of simple peripheral devices. Figure 7.10 illustrates its use to control a keyboard/display terminal. The keyboard provides 8 bits of input. Two of these bits, SHIFT and CONTROL, have special meaning to the keyboard-handling program executing in the processor. However, this interpretation is transparent to the 82C55A, which simply accepts the 8 bits of data and presents them on the system data bus. Two handshaking control lines are provided for use with the keyboard.

The display is also linked by an 8-bit data port. Again, two of the bits have special meanings that are transparent to the 82C55A. In addition to two handshaking lines, two lines provide additional control functions.

7.5 DIRECT MEMORY ACCESS

Drawbacks of Programmed and Interrupt-Driven I/O

Interrupt-driven I/O, though more efficient than simple programmed I/O, still requires the active intervention of the processor to transfer data between memory and an I/O module, and any data transfer must traverse a path through the processor. Thus, both these forms of I/O suffer from two inherent drawbacks:

1. The I/O transfer rate is limited by the speed with which the processor can test and service a device.
2. The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfer (e.g., Figure 7.5).

There is somewhat of a trade-off between these two drawbacks. Consider the transfer of a block of data. Using simple programmed I/O, the processor is dedicated to the task of I/O and can move data at a rather high rate, at the cost of doing nothing else. Interrupt I/O frees up the processor to some extent at the expense of the I/O transfer rate. Nevertheless, both methods have an adverse impact on both processor activity and I/O transfer rate.

When large volumes of data are to be moved, a more efficient technique is required: direct memory access (DMA).

DMA Function

DMA involves an additional module on the system bus. The DMA module (Figure 7.11) is capable of mimicking the processor and, indeed, of taking over control of the system

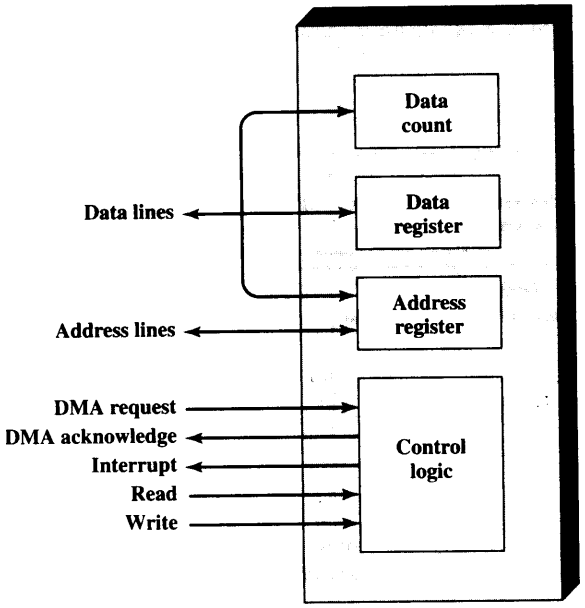


Figure 7.11 Typical DMA Block Diagram

from the processor. It needs to do this to transfer data to and from memory over the system bus. For this purpose, the DMA module must use the bus only when the processor does not need it, or it must force the processor to suspend operation temporarily. The latter technique is more common and is referred to as *cycle stealing*, because the DMA module in effect steals a bus cycle.

When the processor wishes to read or write a block of data, it issues a command to the DMA module, by sending to the DMA module the following information:

- Whether a read or write is requested, using the read or write control line between the processor and the DMA module
- The address of the I/O device involved, communicated on the data lines
- The starting location in memory to read from or write to, communicated on the data lines and stored by the DMA module in its address register
- The number of words to be read or written, again communicated via the data lines and stored in the data count register

The processor then continues with other work. It has delegated this I/O operation to the DMA module. The DMA module transfers the entire block of data, one word at a time, directly to or from memory, without going through the processor. When the transfer is complete, the DMA module sends an interrupt signal to the processor. Thus, the processor is involved only at the beginning and end of the transfer (Figure 7.5c).

Figure 7.12 shows where in the instruction cycle the processor may be suspended. In each case, the processor is suspended just before it needs to use the bus. The DMA module then transfers one word and returns control to the processor. Note that this is not an interrupt; the processor does not save a context and do something else. Rather, the processor pauses for one bus cycle. The overall effect is to cause the processor to execute more slowly. Nevertheless, for a multiple-word I/O transfer, DMA is far more efficient than interrupt-driven or programmed I/O.

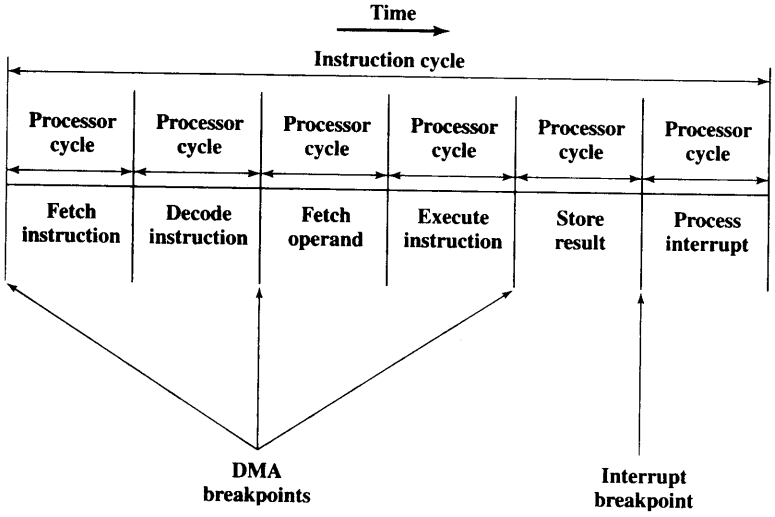
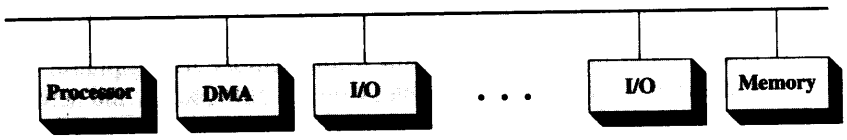


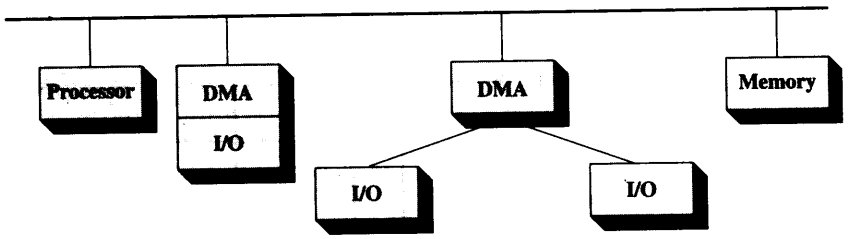
Figure 7.12 DMA and Interrupt Breakpoints during an Instruction Cycle

The DMA mechanism can be configured in a variety of ways. Some possibilities are shown in Figure 7.13. In the first example, all modules share the same system bus. The DMA module, acting as a surrogate processor, uses programmed I/O to exchange data between memory and an I/O module through the DMA module. This configuration, while it may be inexpensive, is clearly inefficient. As with processor-controlled programmed I/O, each transfer of a word consumes two bus cycles.

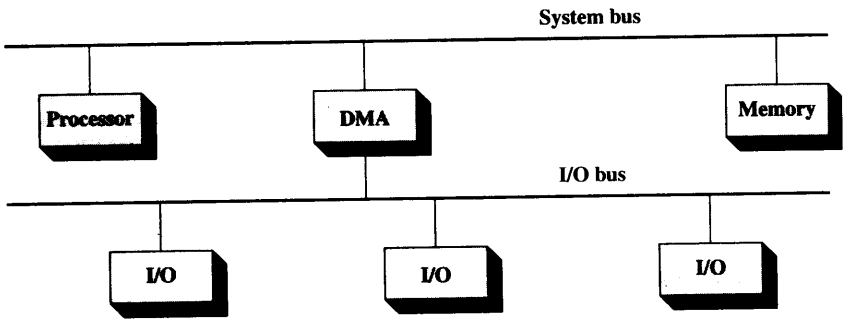
The number of required bus cycles can be cut substantially by integrating the DMA and I/O functions. As Figure 7.13b indicates, this means that there is a path between the DMA module and one or more I/O modules that does not include the system bus. The DMA logic may actually be a part of an I/O module, or it may be a separate module that controls one or more I/O modules. This concept can be taken one step further by connecting I/O modules to the DMA module using an I/O bus (Figure 7.13c). This reduces the number of I/O interfaces in the DMA module to one and provides for an easily expandable configuration. In all of these cases (Figures 7.13b and c), the system bus that the DMA module shares with the processor and memory is used by the DMA module only to exchange data with memory. The exchange of data between the DMA and I/O modules takes place off the system bus.



(a) Single-bus, detached DMA



(b) Single-bus, integrated DMA-I/O



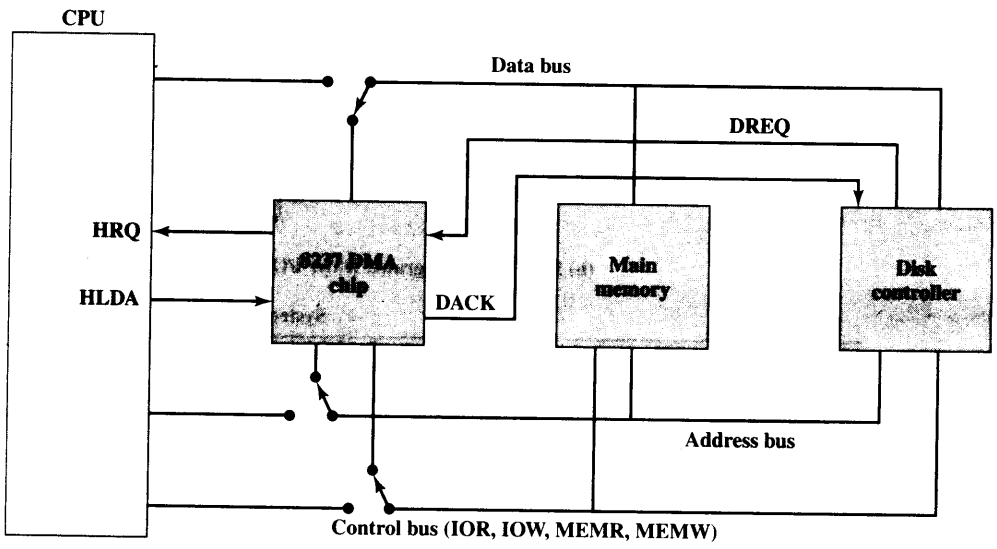
(c) I/O bus

Figure 7.13 Alternative DMA Configurations

Intel 8237A DMA Controller

The Intel 8237A DMA controller interfaces to the 80x86 family of processors and to DRAM memory to provide a DMA capability. Figure 7.14 indicates the location of the DMA module. When the DMA module needs to use the system buses (data, address, and control) to transfer data, it sends a signal called HOLD to the processor. The processor responds with the HLDA (hold acknowledge) signal, indicating that the DMA module can use the buses. For example, if the DMA module is to transfer a block of data from memory to disk, it will do the following:

1. The peripheral device (such as the disk controller) will request the service of DMA by pulling DREQ (DMA request) high.
2. The DMA will put a high on its HRQ (hold request), signaling the CPU through its HOLD pin that it needs to use the buses.
3. The CPU will finish the present bus cycle (not necessarily the present instruction) and respond to the DMA request by putting high on its HDLA (hold acknowledge), thus telling the 8237 DMA that it can go ahead and use the buses to perform its task. HOLD must remain active high as long as DMA is performing its task.
4. DMA will activate DACK (DMA acknowledge), which tells the peripheral device that it will start to transfer the data.
5. DMA starts to transfer the data from memory to peripheral by putting the address of the first byte of the block on the address bus and activating MEMR, thereby reading the byte from memory into the data bus; it then activates IOW to



DACK = DMA acknowledge
DREQ = DMA request
HLDA = HOLD acknowledge
HRQ = HOLD request

Figure 7.14 8237 DMA Usage of System Bus

write it to the peripheral. Then DMA decrements the counter and increments the address pointer and repeats this process until the count reaches zero and the task is finished.

6. After the DMA has finished its job it will deactivate HRQ, signaling the CPU that it can regain control over its buses.

While the DMA is using the buses to transfer data, the processor is idle. Similarly, when the processor is using the bus, the DMA is idle. The 8237 DMA is known as a *fly-by* DMA controller. This means that the data being moved from one location to another does not pass through the DMA chip and is not stored in the DMA chip. Therefore, the DMA can only transfer data between an I/O port and a memory address, but not between two I/O ports or two memory locations. However, as explained subsequently, the DMA chip can perform a memory-to-memory transfer via a register.

The 8237 contains four DMA channels that can be programmed independently and any one of the channels may be active at any moment. These channels are numbered 0, 1, 2, and 3.

The 8237 has a set of five control/command registers to program and control DMA operation over one of its channels (Table 7.4):

- **Command:** The processor loads this register to control the operation of the DMA. D0 enables a memory-to-memory transfer, in which channel 0 is used to transfer a byte into an 8237 temporary register and channel 1 is used to transfer the byte from the register to memory. When memory-to-memory is enabled, D1 can be used to disable increment/decrement on channel 0 so that a fixed value can be written into a block of memory. D2 enables or disables DMA.
- **Status:** The processor reads this register to determine DMA status. Bits D0–D3 are used to indicate if channels 0–3 have reached their TC (terminal count). Bits D4–D7 are used by the processor to determine if any channel has a DMS request pending.
- **Mode:** The processor sets this register to determine the mode of operation of the DMA. Bits D0 and D1 are used to select a channel. The other bits select various operation modes for the selected channel. Bits D2 and D3 determine if the transfer is a from an I/O device to memory (write) or from memory to I/O (read), or a verify operation. If D4 is set, then the memory address register and the count register are reloaded with their original values at the end of a DMA data transfer. Bits D6 and D7 determine the way in which the 8237 is used. In single mode, a single byte of data is transferred. Block and demand modes are used for a block transfer, with the demand mode allowing for premature ending of the transfer. Cascade mode allows multiple 8237s to be cascaded to expand the number of channels to more than 4.
- **Single Mask:** The processor sets this register. Bits D0 and D1 select the channel. Bit D2 clears or sets the mask bit for that channel. It is through this register that the DREQ input of a specific channel can be masked (disabled) or unmasked (enabled). While the command register can be used to disable the whole DMA chip, the single mask register allows the programmer to disable or enable a specific channel.

Table 7.4 Intel 8237A Registers

Bit	Command	Status	Mode	Single Mask	All Mask
D0	Memory-to memory-E/D	Channel 0 has reached TC	Channel select	Select channal mask bit	Clear/set channel 0 mask bit
D1	Channel 0 address hold E/D	Channel 1 has reached TC			Clear/set channel 1 mask bit
D2	Controller E/D	Channel 2 has reached TC	Verify/write/read transfer	Clear/set mask bit	Clear/set channel 2 mask bit
D3	Normal/compressed timing	Channel 3 has reached TC		Not used	Clear/set channel 3 mask bit
D4	Fixed/rotating priority	Channel 0 request	Autoinitialization E/D		Not used
D5	Late/extended write selection	Channel 0 request	Address increment/decrement select		
D6	DREQ sense active high/low	Channel 0 request	Demand/single/block/ cascade mode select		
D7	DACK sense active high/low	Channel 0 request			

E/D = enable/disable
TC = terminal count

- **All Mask:** This register is similar to the single mask register except that all 4 channels can be masked or unmasked with one write operation.

In addition, the 8237A has eight data registers: one memory address register and one count register for each channel. The processor sets these registers to indicate the location of size of main memory to be affected by the transfers.

7.6 I/O CHANNELS AND PROCESSORS

The Evolution of the I/O Function

As computer systems have evolved, there has been a pattern of increasing complexity and sophistication of individual components. Nowhere is this more evident than in the I/O function. We have already seen part of that evolution. The evolutionary steps can be summarized as follows:

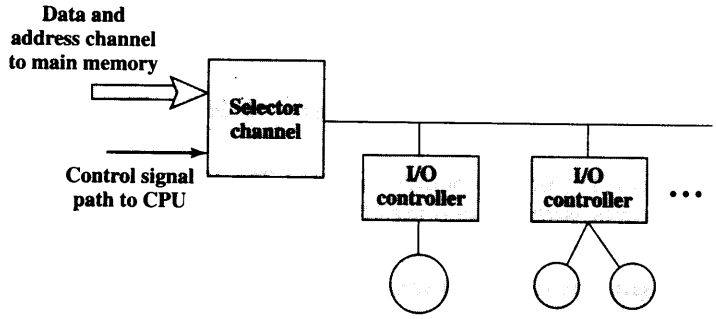
1. The CPU directly controls a peripheral device. This is seen in simple microprocessor-controlled devices.
2. A controller or I/O module is added. The CPU uses programmed I/O without interrupts. With this step, the CPU becomes somewhat divorced from the specific details of external device interfaces.
3. The same configuration as in step 2 is used, but now interrupts are employed. The CPU need not spend time waiting for an I/O operation to be performed, increasing efficiency.
4. The I/O module is given direct access to memory via DMA. It can now move a block of data to or from memory without involving the CPU, except at the beginning and end of the transfer.
5. The I/O module is enhanced to become a processor in its own right, with a specialized instruction set tailored for I/O. The CPU directs the I/O processor to execute an I/O program in memory. The I/O processor fetches and executes these instructions without CPU intervention. This allows the CPU to specify a sequence of I/O activities and to be interrupted only when the entire sequence has been performed.
6. The I/O module has a local memory of its own and is, in fact, a computer in its own right. With this architecture, a large set of I/O devices can be controlled, with minimal CPU involvement. A common use for such an architecture has been to control communication with interactive terminals. The I/O processor takes care of most of the tasks involved in controlling the terminals.

As one proceeds along this evolutionary path, more and more of the I/O function is performed without CPU involvement. The CPU is increasingly relieved of I/O-related tasks, improving performance. With the last two steps (5–6), a major change occurs with the introduction of the concept of an I/O module capable of executing a program. For step 5, the I/O module is often referred to as an *I/O channel.* For step 6, the term *I/O processor* is often used. However, both terms are on occasion applied to both situations. In what follows, we will use the term *I/O channel.*

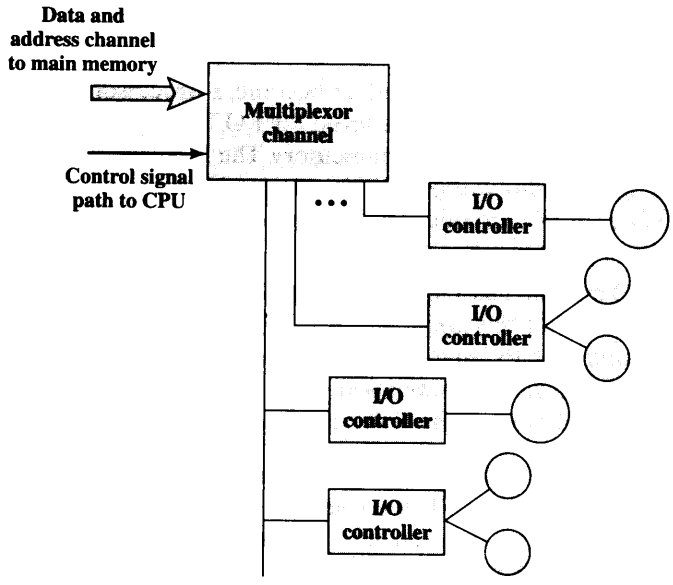
Characteristics of I/O Channels

The I/O channel represents an extension of the DMA concept. An I/O channel has the ability to execute I/O instructions, which gives it complete control over I/O operations. In a computer system with such devices, the CPU does not execute I/O instructions. Such instructions are stored in main memory to be executed by a special-purpose processor in the I/O channel itself. Thus, the CPU initiates an I/O transfer by instructing the I/O channel to execute a program in memory. The program will specify the device or devices, the area or areas of memory for storage, priority, and actions to be taken for certain error conditions. The I/O channel follows these instructions and controls the data transfer.

Two types of I/O channels are common, as illustrated in Figure 7.15. A *selector channel* controls multiple high-speed devices and, at any one time, is dedi-



(a) Selector



(b) Multiplexor

Figure 7.15 I/O Channel Architecture

cated to the transfer of data with one of those devices. Thus, the I/O channel selects one device and effects the data transfer. Each device, or a small set of devices, is handled by a *controller*, or I/O module, that is much like the I/O modules we have been discussing. Thus, the I/O channel serves in place of the CPU in controlling these I/O controllers. A *multiplexor channel* can handle I/O with multiple devices at the same time. For low-speed devices, a *byte multiplexor* accepts or transmits characters as fast as possible to multiple devices. For example, the resultant character stream from three devices with different rates and individual streams $A_1A_2A_3A_4\ldots$, $B_1B_2B_3B_4\ldots$, and $C_1C_2C_3C_4\ldots$ might be $A_1B_1C_1A_2C_2A_3B_2C_3A_4$, and so on. For high-speed devices, a *block multiplexor* interleaves blocks of data from several devices.

7.7 THE EXTERNAL INTERFACE: FIREWIRE AND INFINIBAND

Types of Interfaces

The interface to a peripheral from an I/O module must be tailored to the nature and operation of the peripheral. One major characteristic of the interface is whether it is serial or parallel (Figure 7.16). In a **parallel interface**, there are multiple lines connecting the I/O module and the peripheral, and multiple bits are transferred simultaneously, just as all of the bits of a word are transferred simultaneously over the data bus. In a **serial interface**, there is only one line used to transmit data, and bits must be transmitted one at a time. A parallel interface has traditionally been used for higher-speed peripherals, such as tape and disk, while the serial interface has traditionally been used for printers and terminals. With a new generation of high-speed serial interfaces, parallel interfaces are becoming much less common.

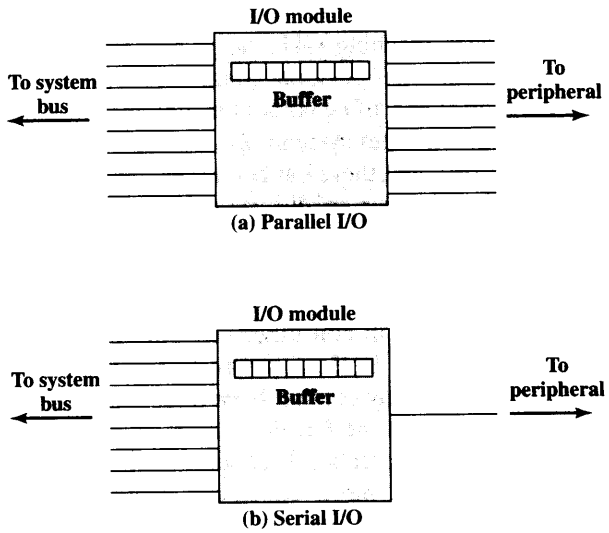


Figure 7.16 Parallel and Serial I/O

In either case, the I/O module must engage in a dialogue with the peripheral. In general terms, the dialogue for a write operation is as follows:

1. The I/O module sends a control signal requesting permission to send data.
2. The peripheral acknowledges the request.
3. The I/O module transfers data (one word or a block depending on the peripheral).
4. The peripheral acknowledges receipt of the data.

A read operation proceeds similarly.

Key to the operation of an I/O module is an internal buffer that can store data being passed between the peripheral and the rest of the system. This buffer allows the I/O module to compensate for the differences in speed between the system bus and its external lines.

Point-to-Point and Multipoint Configurations

The connection between an I/O module in a computer system and external devices can be either point-to-point or multipoint. A point-to-point interface provides a dedicated line between the I/O module and the external device. On small systems (PCs, workstations), typical point-to-point links include those to the keyboard, printer, and external modem. A typical example of such an interface is the EIA-232 specification (see [STAL04] for a description).

Of increasing importance are multipoint external interfaces, used to support external mass storage devices (disk and tape drives) and multimedia devices (CD-ROMs, video, audio). These multipoint interfaces are in effect external buses, and they exhibit the same type of logic as the buses discussed in Chapter 3. In this section, we look at two key examples: FireWire and InfiniBand.

FireWire Serial Bus

With processor speeds reaching GHz range and storage devices holding multiple gigabits, the I/O demands for personal computers, workstations, and servers are formidable. Yet the high-speed I/O channel technologies that have been developed for mainframe and supercomputer systems are too expensive and bulky for use on these smaller systems. Accordingly, there has been great interest in developing a high-speed alternative to SCSI and other small-system I/O interfaces. The result is the IEEE standard 1394, for a High Performance Serial Bus, commonly known as FireWire.

FireWire has a number of advantages over older I/O interfaces. It is very high speed, low cost, and easy to implement. In fact, FireWire is finding favor not only for computer systems, but also in consumer electronics products, such as digital cameras, VCRs, and televisions. In these products, FireWire is used to transport video images, which are increasingly coming from digitized sources.

One of the strengths of the FireWire interface is that it uses serial transmission (bit at a time) rather than parallel. Parallel interfaces, such as SCSI, require more wires, which means wider, more expensive cables and wider, more expensive connectors with more pins to bend or break. A cable with more wires requires shielding to prevent electrical interference between the wires. Also, with a parallel

interface, synchronization between wires becomes a requirement, a problem that gets worse with increased cable length.

In addition, computers are getting physically smaller even as they expand in computing power and I/O needs. Handheld and pocket-size computers have little room for connectors yet need high data rates to handle images and video.

The intent of FireWire is to provide a single I/O interface with a simple connector that can handle numerous devices through a single port, so that the mouse, laser printer, external disk drive, sound, and local area network hookups can be replaced with this single connector. The connector is inspired by the one used in the Nintendo Gameboy. It is so convenient that the user can reach behind the machine and plug it in without looking.

FireWire Configurations FireWire uses a daisy-chain configuration, with up to 63 devices connected off a single port. Moreover, up to 1022 FireWire buses can be interconnected using bridges, enabling a system to support as many peripherals as required.

FireWire provides for what is known as hot plugging, which makes it possible to connect and disconnect peripherals without having to power the computer system down or reconfigure the system. Also, FireWire provides for automatic configuration; it is not necessary manually to set device Ids or to be concerned with the relative position of devices. Figure 7.17 shows a simple FireWire configuration. With FireWire, there are no terminations, and the system automatically performs a configuration function to assign addresses. Also note that a FireWire bus need not be a strict daisy chain. Rather, a tree-structured configuration is possible.

An important feature of the FireWire standard is that it specifies a set of three layers of protocols to standardize the way in which the host system interacts with the peripheral devices over the serial bus. Figure 7.18 illustrates this stack. The three layers of the stack are

- **Physical layer:** Defines the transmission media that are permissible under FireWire and the electrical and signaling characteristics of each
- **Link layer:** Describes the transmission of data in the packets
- **Transaction layer:** Defines a request–response protocol that hides the lower-layer details of FireWire from applications

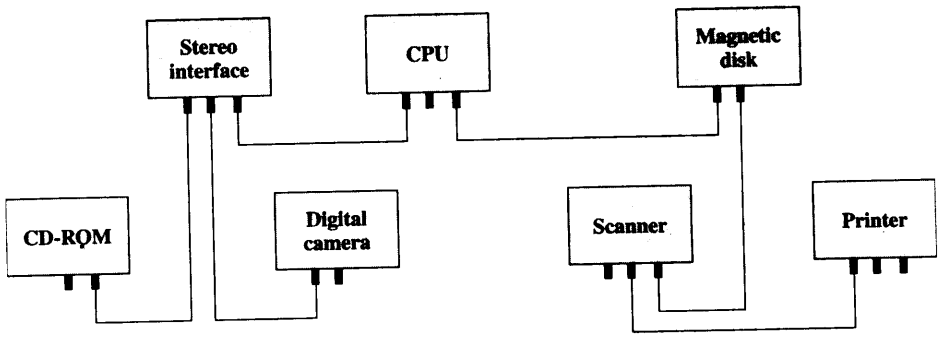


Figure 7.17 Simple FireWire Configuration

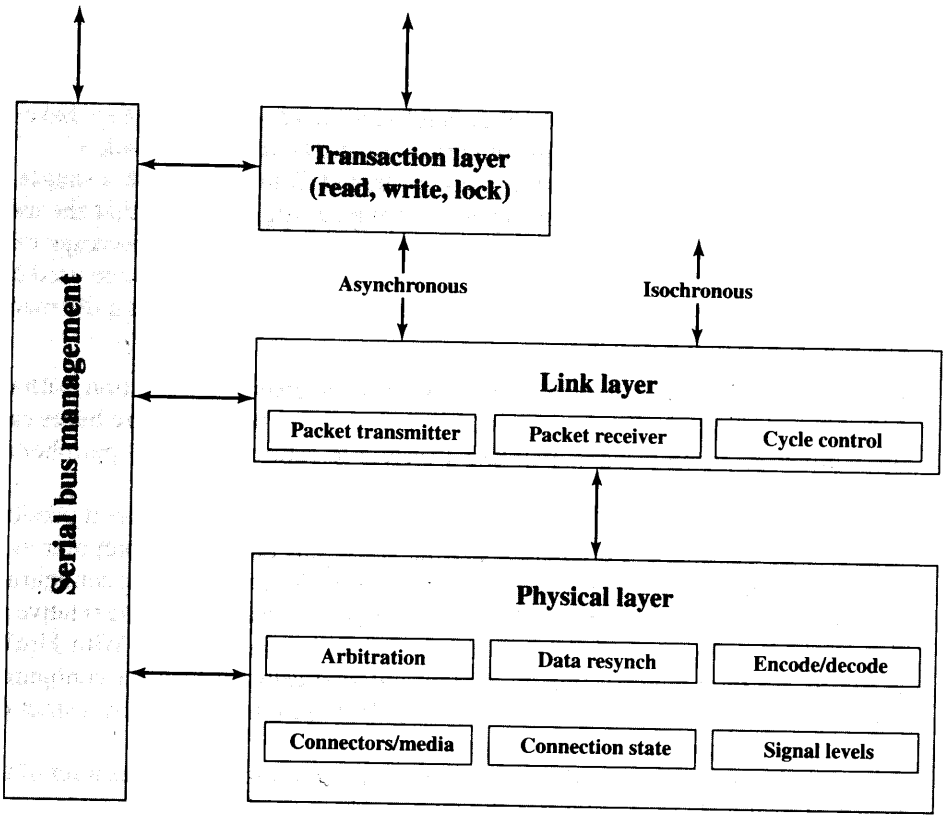


Figure 7.18 FireWire Protocol Stack

Physical Layer The physical layer of FireWire specifies several alternative transmission media and their connectors, with different physical and data transmission properties. Data rates from 25 to 400 Mbps are defined. The physical layer converts binary data into electrical signals for various physical media. This layer also provides the arbitration service that guarantees that only one device at a time will transmit data.

Two forms of arbitration are provided by FireWire. The simplest form is based on the tree-structured arrangement of the nodes on a FireWire bus, mentioned earlier. A special case of this structure is a linear daisy chain. The physical layer contains logic that allows all the attached devices to configure themselves so that one node is designated as the root of the tree and other nodes are organized in a parent/child relationship forming the tree topology. Once this configuration is established, the root node acts as a central arbiter and processes requests for bus access in a first-come-first-served fashion. In the case of simultaneous requests, the node with the highest natural priority is granted access. The natural priority is determined by which competing node is closest to the root and, among those of equal distance from the root, which one has the lower ID number.

The aforementioned arbitration method is supplemented by two additional functions: fair arbitration and urgent arbitration. With fairness arbitration, time on the bus is organized into *fairness intervals*. At the beginning of an interval, each node

sets an arbitration_enable flag. During the interval, each node may compete for bus access. Once a node has gained access to the bus, it resets its arbitration_enable flag and may not again compete for fair access during this interval. This scheme makes the arbitration fairer, in that it prevents one or more busy high-priority devices from monopolizing the bus.

In addition to the fairness scheme, some devices may be configured as having *urgent* priority. Such nodes may gain control of the bus multiple times during a fairness interval. In essence, a counter is used at each high-priority node that enables the high-priority nodes to control 75% of the available bus time. For each packet that is transmitted as nonurgent, three packets may be transmitted as urgent.

Link Layer The link layer defines the transmission of data in the form of packets. Two types of transmission are supported:

- **Asynchronous:** A variable amount of data and several bytes of transaction layer information are transferred as a packet to an explicit address and an acknowledgment is returned.
- **Isochronous:** A variable amount of data is transferred in a sequence of fixed-size packets transmitted at regular intervals. This form of transmission uses simplified addressing and no acknowledgment.

Asynchronous transmission is used by data that have no fixed data rate requirements. Both the fair arbitration and urgent arbitration schemes may be used for asynchronous transmission. The default method is fair arbitration. Devices that desire a substantial fraction of the bus capacity or have severe latency requirements use the urgent arbitration method. For example, a high-speed real-time data collection node may use urgent arbitration when critical data buffers are more than half full.

Figure 7.19a depicts a typical asynchronous transaction. The process of delivering a single packet is called a subaction. The subaction consists of five time periods:

- **Arbitration sequence:** This is the exchange of signals required to give one device control of the bus.
- **Packet transmission:** Every packet includes a header containing the source and destination Ids. The header also contains packet type information, a CRC (cyclic redundancy check) checksum, and parameter information for the specific packet type. A packet may also include a data block consisting of user data and another CRC.
- **Acknowledgment gap:** This is the time delay for the destination to receive and decode a packet and generate an acknowledgment.
- **Acknowledgment:** The recipient of the packet returns an acknowledgment packet with a code indicating the action taken by the recipient.
- **Subaction gap:** This is an enforced idle period to ensure that other nodes on the bus do not begin arbitrating before the acknowledgment packet has been transmitted.

At the time that the acknowledgment is sent, the acknowledging node is in control of the bus. Therefore, if the exchange is a request/response interaction

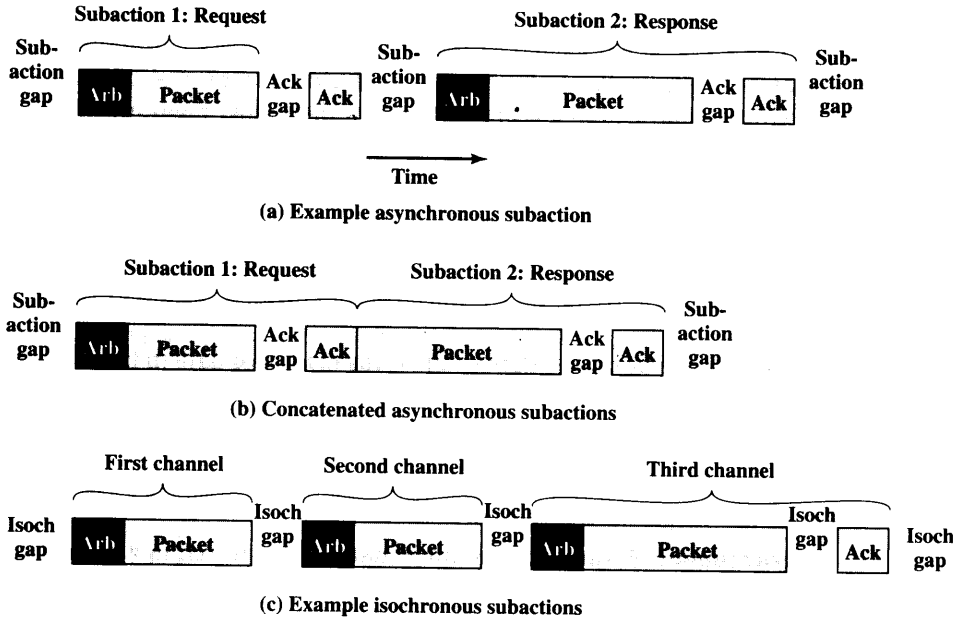


Figure 7.19 FireWire Subactions

between two nodes, then the responding node can immediately transmit the response packet without going through an arbitration sequence (Figure 7.19b).

For devices that regularly generate or consume data, such as digital sound or video, isochronous access is provided. This method guarantees that data can be delivered within a specified latency with a guaranteed data rate.

To accommodate a mixed traffic load of isochronous and asynchronous data sources, one node is designated as *cycle master.* Periodically, the cycle master issues a cycle_start packet. This signals all other nodes that an isochronous cycle has begun. During this cycle, only isochronous packets may be sent (Figure 7.19c). Each isochronous data source arbitrates for bus access. The winning node immediately transmits a packet. There is no acknowledgment to this packet, and so other isochronous data sources immediately arbitrate for the bus after the previous isochronous packet is transmitted. The result is that there is a small gap between the transmission of one packet and the arbitration period for the next packet, dictated by delays on the bus. This delay, referred to as the isochronous gap, is smaller than a subaction gap.

After all isochronous sources have transmitted, the bus will remain idle long enough for a subaction gap to occur. This is the signal to the asynchronous sources that they may now compete for bus access. Asynchronous sources may then use the bus until the beginning of the next isochronous cycle.

Isochronous packets are labeled with 8-bit channel numbers that are previously assigned by a dialogue between the two nodes that are to exchange isochronous data. The header, which is shorter than that for asynchronous packets, also includes a data length field and a header CRC.

InfiniBand

InfiniBand is a recent I/O specification aimed at the high-end server market.[4] The first version of the specification was released in early 2001 and has attracted numerous vendors. The standard describes an architecture and specifications for data flow between processors and intelligent I/O devices. InfiniBand is intended to replace the PCI bus in servers, to provide greater capacity, increased expandability, and enhanced flexibility in server design. In essence, InfiniBand enables servers, remote storage, and other network devices to be attached in a central fabric of switches and links. The switch-based architecture can connect up to 64,000 servers, storage systems, and networking devices.

InfiniBand Architecture Although PCI is a reliable interconnect method and continues to provide increased speeds, up to 1 Gbps, it is a limited architecture compared to InfiniBand. With InfiniBand, it is not necessary to have the basic I/O interface hardware inside the server chassis. With InfiniBand, remote storage, networking, and connections between servers are accomplished by attaching all devices to a central fabric of switches and links. Removing I/O from the server chassis allows greater server density and allows for a more flexible and scalable data center, as independent nodes may be added as needed.

Unlike PCI, which measures distances from a CPU motherboard in centimeters, InfiniBand's channel design enables I/O devices to be placed up to 17 meters away from the server using copper, up to 300 m using multimode optical fiber, and up to 10 km with single-mode optical fiber. Transmission rates has high as 30 Gbps can be achieved.

Figure 7.20 illustrates the InfiniBand architecture. The key elements are as follows:

- **Host channel adapter (HCA):** Instead of a number of PCI slots, a typical server needs a single interface to an HCA that links the server to an InfiniBand switch. The HCA attaches to the server at a memory controller, which has access to the system bus and controls traffic between the processor and memory and between the HCA and memory. The HCA uses direct-memory access (DMA) to read and write memory.
- **Target channel adapter (TCA):** A TCA is used to connect storage systems, routers, and other peripheral devices to an InfiniBand switch.
- **InfiniBand switch:** A switch provides point-to-point physical connections to a variety of devices and switches traffic from one link to another. Servers and devices communicate through their adapters, via the switch. The switch's intelligence manages the linkage without interrupting the servers' operation.
- **Links:** The link between a switch and a channel adapter, or between two switches.
- **Subnet:** A subnet consists of one or more interconnected switches plus the links that connect other devices to those switches. Figure 7.20 shows a subnet with a single switch, but more complex subnets are required when a large number of devices are to be interconnected. Subnets allow administrators to confine broadcast and multicast transmissions within the subnet.

[4]InfiniBand is the result of the merger of two competing projects: Future I/O (backed by Cisco, HP, Compaq, and IBM) and Next Generation I/O (developed by Intel and backed by a number of other companies).

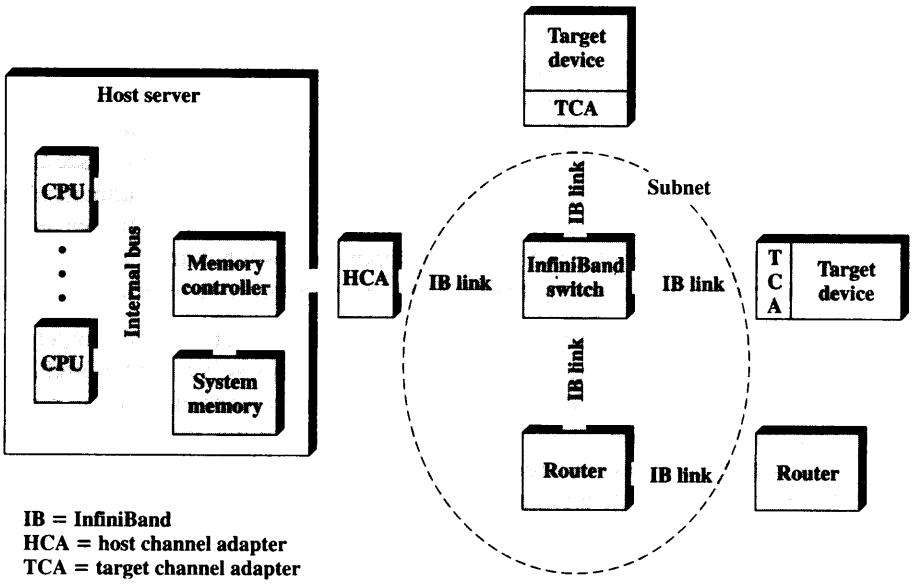


Figure 7.20 InfiniBand Switch Fabric

- **Router:** Connects InfiniBand subnets, or connects an InfiniBand switch to a network, such as a local area network, wide area network, or storage area network.

The channel adapters are intelligent devices that handle all I/O functions without the need to interrupt the server's processor. For example, there is a control protocol by which a switch discovers all TCAs and HCAs in the fabric and assigns logical addresses to each. This is done without processor involvement.

The InfiniBand switch temporarily opens up channels between the processor and devices with which it is communicating. The devices do not have to share a channel's capacity, as is the case with a bus-based design such as PCI, which requires that devices arbitrate for access to the processor. Additional devices are added to the configuration by hooking up each device's TCA to the switch.

InfiniBand Operation Each physical link between a switch and an attached interface (HCA or TCA) can be support up to 16 logical channels, called **virtual lanes**. One lane is reserved for fabric management and the other lanes for data transport. Data are sent in the form of a stream of packets, with each packet containing some portion of the total data to be transferred, plus addressing and control information. Thus, a set of communications protocols are used to manage the transfer of data. A virtual lane is temporarily dedicated to the transfer of data from one end node to another over the InfiniBand fabric. The InfiniBand switch maps traffic from an incoming lane to an outgoing lane to route the data between the desired end points.

Figure 7.21 indicates the logical structure used to support exchanges over InfiniBand. To account for the fact that some devices can send data faster than another destination device can receive it, a pair of queues at both ends of each link temporarily buffers excess outbound and inbound data. The queues can be located in the channel adapter or in the attached device's memory. A separate pair of queues is

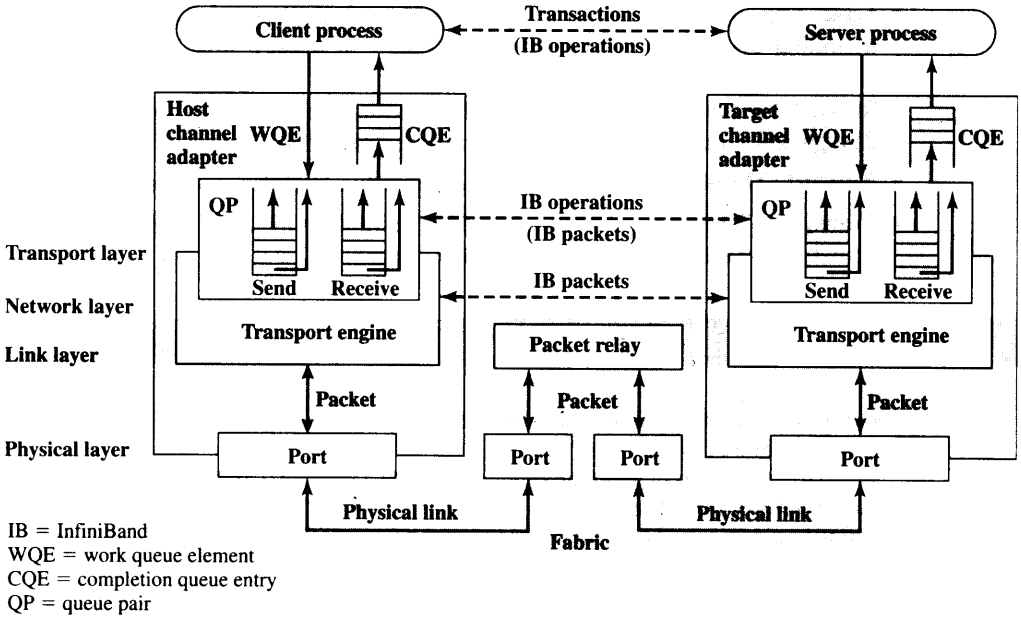


Figure 7.21 InfiniBand Communication Protocol Stack

used for each virtual lane. The host uses these queues in the following fashion. The host places a transaction, called a work queue entry (WQE) into either the send or receive queue of the queue pair. The two most important WQEs are SEND and RECEIVE. For a SEND operation, the WQE specifies a block of data in the device's memory space for the hardware to send to the destination. A RECEIVE WQE specifies where the hardware is to place data received from another device when that consumer executes a SEND operation. The channel adapter processes each posted WQE in the proper prioritized order and generates a completion queue entry (CQE) to indicate the completion status.

Figure 7.21 also indicates that a layered protocol architecture is used, consisting of four layers:

- **Physical:** The physical-layer specification defines three link speeds (1X, 4X, and 12X) giving transmission rates of 2.5, 10, and 30 Gbps, respectively (Table 7.5). The physical layer also defines the physical media, including copper and optical fiber.

Table 7.5 InfiniBand Links and Data Throughput Rates

Link	Signal rate (unidirectional)	Usable capacity (80% of signal rate)	Effective data throughput (send + receive)
1-wide	2.5 Gbps	2 Gbps (250 MBps)	(250 + 250) MBps
4-wide	10 Gbps	8 Gbps (1 GBps)	(1 + 1) GBps
12-wide	30 Gbps	24 Gbps (3 GBps)	(3 + 3) Gbps

- **Link:** This layer defines the basic packet structure used to exchange data, including an addressing scheme that assigns a unique link address to every device in a subnet. This level includes the logic for setting up virtual lanes and for switching data through switches from source to destination within a subnet. The packet structure includes an error detection code to provide reliability.
- **Network:** The network layer routes packets between different InfiniBand subnets.
- **Transport:** The transport layer provides reliability mechanism for end-to-end transfer of packets across one or more subnets.

7.8 RECOMMENDED READING AND WEB SITES

A good discussion of Intel I/O modules and architecture, including the 82C59A, 82C55A, and 8237A can be found in [MAZI03].

FireWire is covered in great detail in [ANDE98]. [WICK97] and [THOM00] provide a concise overviews of FireWire.

InfiniBand is covered in great detail in [SHAN03] and [FUTR01]. [KAGA01] provides a concise overview.

ANDE98 Anderson, D. *FireWire System Architecture.* Reading, MA: Addison-Wesley, 1998.

FUTR01 Futral, W. *InfiniBand Architecture: Development and Deployment.* Hillsboro, OR: Intel Press, 2001.

KAGA01 Kagan, M. "InfiniBand: Thinking Outside the Box Design." *Communications System Design*, September 2001. (**www.csdmag.com**)

MAZI03 Mazidi, M., and Mazidi, J. *The 80x86 IBM PC and Compatible Computers: Assembly Language, Design and Interfacing.* Upper Saddle River, NJ: Prentice Hall, 2003.

SHAN03 Shanley, T. *InfinBand Network Architecture.* Reading, MA: Addison-Wesley, 2003.

THOM00 Thompson, D. "IEEE 1394: Changing the Way We Do Multimedia Communications." *IEEE Multimedia*, April–June 2000.

WICK97 Wickelgren, I. "The Facts About FireWire." *IEEE Spectrum*, April 1997.

Recommended Web Sites:

- **T10 Home Page:** T10 is a Technical Committee of the National Committee on Information Technology Standards and is responsible for lower-level interfaces. Its principal work is the Small Computer System Interface (SCSI).
- **1394 Trade Association:** Includes technical information and vendor pointers on FireWire.
- **InfiniBand Trade Association:** Includes technical information and vendor pointers on InfiniBand.
- **I/O Characterization and Optimization:** A facility dedicated to education and research in the area of I/O design and performance. Useful tools and tutorials. Operated by the University of Illinois.

7.9 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

cycle stealing	I/O channel	multiplexor channel
direct memory access (DMA)	I/O command	parallel I/O
FireWire	I/O module	peripheral device
InfiniBand	I/O processor	programmed I/O
interrupt	isolated I/O	selector channel
interrupt-driven I/O	memory-mapped I/O	serial I/O

Review Questions

7.1 List three broad classifications of external, or peripheral, devices.

7.2 What is the International Reference Alphabet?

7.3 What are the major functions of an I/O module?

7.4 List and briefly define three techniques for performing I/O.

7.5 What is the difference between memory-mapped I/O and isolated I/O?

7.6 When a device interrupt occurs, how does the processor determine which device issued the interrupt?

7.7 When a DMA module takes control of a bus, and while it retains control of the bus, what does the processor do?

Problems

7.1 On a typical microprocessor, a distinct I/O address is used to refer to the I/O data registers and a distinct address for the control and status registers in an I/O controller for a given device. Such registers are referred to as **ports**. In the Intel 8088, two I/O instruction formats are used. In one format, the 8-bit opcode specifies an I/O operation; this is followed by an 8-bit port address. Other I/O opcodes imply that the port address is in the 16-bit DX register. How many ports can the 8088 address in each I/O addressing mode?

7.2 A similar instruction format is used in the Zilog Z8000 microprocessor family. In this case, there is a direct port addressing capability, in which a 16-bit port address is part of the instruction, and an indirect port addressing capability, in which the instruction references one of the 16-bit general purpose registers, which contains the port address. How many ports can the Z8000 address in each I/O addressing mode?

7.3 The Z8000 also includes a block I/O transfer capability that, unlike DMA, is under the direct control of the processor. The block transfer instructions specify a port address register (Rp), a count register (Rc) and a destination register (Rd). Rd contains the main memory address at which the first byte read from the input port is to be stored. Rc is any of the 16-bit general purpose registers. How large a data block can be transferred?

7.4 Consider a microprocessor that has a block I/O transfer instruction such as that found on the Z8000. Following its first execution, such an instruction takes five clock cycles to reexecute. However, if we employ a non-block I/O instruction, it takes a total of 20 clock cycles for fetching and execution. Calculate the increase in speed with the block I/O instruction when transferring blocks of 128 bytes.

7.5 A system is based on an 8-bit microprocessor and has two I/O devices. The I/O controllers for this system use separate control and status registers. Both devices handle data on a one-byte-at-a-time basis. The first device has two status lines and three control lines. The second device has three status lines and four control lines.

a. How many 8-bit I/O control module registers do we need for status reading and control of each device?

b. What is the total number of needed control module registers given that the first device is an output-only device?

c. How many distinct addresses are needed to control the two devices?

7.6 For programmed I/O, Figure 7.5 indicates that the processor is stuck in a wait loop doing status checking of an I/O device. To increase efficiency, the I/O software could be written so that the processor periodically checks the status of the device. If the device is not ready, the processor can jump to other tasks. After some timed interval, the processor comes back to check status again.

a. Consider the above scheme for outputting data one character at a time to a printer that operates at 10 characters per second (cps). What will happen if its status is scanned every 200 ms?

b. Next consider a keyboard with a single character buffer. On average, characters are entered at a rate of 10 cps. However, the time interval between two consecutive key depressions can be as short as 60 ms. At what frequency should the keyboard be scanned by the I/O program?

7.7 A microprocessor scans the status of an output I/O device every 20 ms. This is accomplished by means of a timer alerting the processor every 20 ms. The interface of the device includes two ports: one for status and one for data output. How long does it take to scan and service the device given a clocking rate of 8 MHz? Assume for simplicity that all pertinent instruction cycles take 12 clock cycles.

7.8 In Section 7.3, one advantage and one disadvantage of memory-mapped I/O, compared with isolated I/O, were listed. List two more advantages and two more disadvantages.

7.9 A particular system is controlled by an operator through commands entered from a keyboard. The average number of commands entered in an 8-hour interval is 60.

a. Suppose the processor scans the keyboard every 100 ms. How many times will the keyboard be checked in an 8-hour period?

b. By what fraction would the number of processor visits to the keyboard be reduced if interrupt-driven I/O were used?

7.10 Consider a system employing interrupt-driven I/O for a particular device that transfers data at an average of 8 KB/s on a continuous basis.

a. Assume that interrupt processing takes about 100 μs (i.e., the time to jump to the interrupt service routine (ISR), execute it, and return to the main program). Determine what fraction of processor time is consumed by this I/O device if it interrupts for every byte.

b. Now assume that the device has two 16-byte buffers and interrupts the processor when one of the buffers is full. Naturally, interrupt processing takes longer, because the ISR must transfer 16 bytes. While executing the ISR, the processor takes about 8 μs for the transfer of each byte. Determine what fraction of processor time is consumed by this I/O device in this case.

c. Now assume that the processor is equipped with a block transfer I/O instruction such as that found on the Z8000. This permits the associated ISR to transfer each byte of a block in only 2 μs. Determine what fraction of processor time is consumed by this I/O device in this case.

7.11 In virtually all systems that include DMA modules, DMA access to main memory is given higher priority than CPU access to main memory. Why?

7.12 A DMA module is transferring characters to memory using cycle stealing, from a device transmitting at 9600 bps. The processor is fetching instructions at the rate of 1 million instructions per second (1 MIPS). By how much will the processor be slowed down due to the DMA activity?

7.13 Consider a system in which a data transfer over a bus takes 500 ns. Transfer of bus control in either direction, from processor to I/O device or vice versa, takes 250 ns. One of the I/O devices has a data transfer rate of 50 KB/s and employs DMA. Data are transferred one byte at a time.

a. Suppose we employ DMA in a burst mode. That is, the DMA interface gains bus mastership prior to the start of a block transfer and maintains control of the bus until the whole block is transferred. For how long would the device tie up the bus when transferring a block of 128 bytes?

b. Repeat the calculation for cycle-stealing mode.

7.14 Examination of the timing diagram of the 8237A indicates that once a block transfer begins, it takes three bus clock cycles per DMA cycle. During the DMA cycle, the 8237A transfers one byte of information between memory and I/O device.

a. Suppose we clock the 8237A at a rate of 5 MHz. How long does it take to transfer one byte?

b. What would be the maximum attainable data transfer rate?

c. Assume that the memory is not fast enough and we have to insert two wait states per DMA cycle. What will be the actual data transfer rate?

7.15 Assume that in the system of the preceding problem, a memory cycle takes 750 ns. To what value could we reduce the clocking rate of the bus without effect on the attainable data transfer rate?

7.16 A DMA controller servers four receive-only telecommunication links (one per DMA channel) having a speed of 64 Kbps each.

a. Would you operate the controller in burst mode or in cycle-stealing mode?

b. What priority scheme would you employ for service of the DMA channels?

7.17 A 32-bit computer has two selector channels and one multiplexor channel. Each selector channel supports two magnetic disk and two magnetic tape units. The multiplexor channel has two line printers, two card readers, and 10 VDT terminals connected to it. Assume the following transfer rates:

Disk drive	800 KBytes/s
Magnetic tape drive	200 KBytes/s
Line printer	6.6 KBytes/s
Card reader	1.2 KBytes/s
VDT	1 KBytes/s

Estimate the maximum aggregate I/O transfer rate in this system.

7.18 A computer consists of a processor and an I/O device D connected to main memory M via a shared bus with a data bus width of one word. The processor can execute a maximum of 10^6 instructions per second. An average instruction requires five machine cycles, three of which use the memory bus. A memory read or write operation uses one machine cycle. Suppose that the processor is continuously executing "background" programs that require 95% of its instruction execution rate but not any I/O instructions. Assume that one processor cycle equals one bus cycle. Now suppose the I/O device is to be used to transfer very large blocks of data between M and D.

a. If programmed I/O is used and each one-word I/O transfer requires the processor to execute two instructions, estimate the maximum I/O data-transfer rate, in words per second, possible through D.

b. Estimate the same rate if DMA is used.

7.19 A data source produces 7-bit IRA characters, to each of which is appended a parity bit. Derive an expression for the maximum effective data rate (rate of IRA data bits) over an R-bps line for the following:

a. Asynchronous transmission, with a 1.5-unit stop bit

b. Bit-synchronous transmission, with a frame consisting of 48 control bits and 128 information bits

c. Same as (b), with a 1024-bit information field

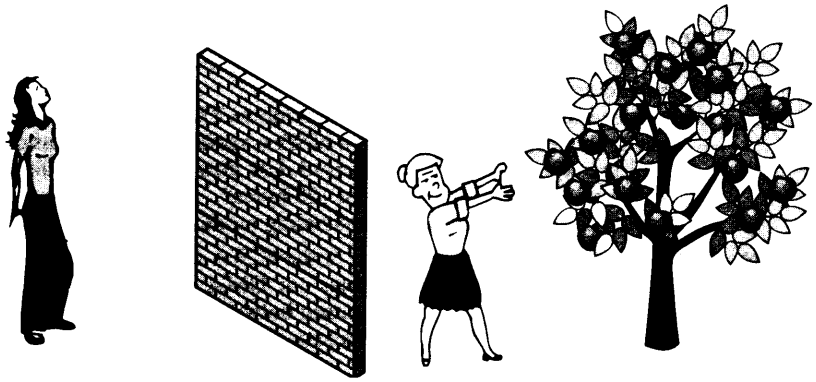


Figure 7.22 An Apple Problem

d. Character-synchronous, with 9 control characters per frame and 16 information characters

e. Same as (d), with 128 information characters

7.20 The following problem is based on a suggested illustration of I/O mechanisms in [ECKE90] (Figure 7.22):

Two women are on either side of a high fence. One of the women, named Apple-server, has a beautiful apple tree loaded with delicious apples growing on her side of the fence; she is happy to supply apples to the other woman whenever needed. The other woman, named Apple-eater, loves to eat apples but has none. In fact, she must eat his apples at a fixed rate (an apple a day keeps the doctor away). If she eats them faster than that rate, she will get sick. If she eats them slower, she will suffer malnutrition. Neither woman can talk, and so the problem is to get apples from Apple-server to Apple-eater at the correct rate.

a. Assume that there is an alarm clock sitting on top of the fence and that the clock can have multiple alarm settings. How can the clock be used to solve the problem? Draw a timing diagram to illustrate the solution.

b. Now assume that there is no alarm clock. Instead Apple-eater has a flag that she can wave whenever she needs an apple. Suggest a new solution. Would it be helpful for Apple-server also to have a flag? If so, incorporate this into the solution. Discuss the drawbacks of this approach.

c. Now take away the flag and assume the existence of a long piece of string. Suggest a solution that is superior to that of (b) using the string.

7.21 Assume that one 16-bit and two 8-bit microprocessors are to be interfaced to a system bus. The following details are given:

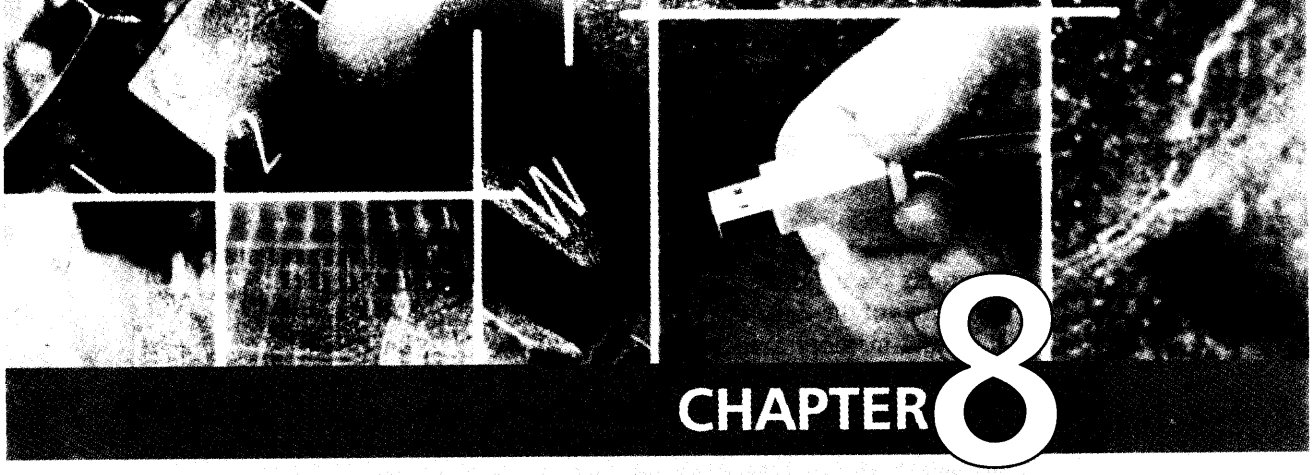
1. All microprocessors have the hardware features necessary for any type of data transfer: programmed I/O, interrupt-driven I/O, and DMA.
2. All microprocessors have a 16-bit address bus.
3. Two memory boards, each of 64 KBytes capacity, are interfaced with the bus. The designer wishes to use a shared memory that is as large as possible.
4. The system bus supports a maximum of four interrupt lines and one DMA line.

Make any other assumptions necessary, and

a. Give the system bus specifications in terms of number and types of lines.

b. Describe a possible protocol for communicating on the bus (i.e., read/write, interrupt, and DMA sequences).

c. Explain how the aforementioned devices are interfaced to the system bus.



CHAPTER 8

OPERATING SYSTEM SUPPORT

KEY POINTS

- The operating system (OS) is the software that controls the execution of programs on a processor and that manages the processor's resources. A number of the functions performed by the OS, including process scheduling and memory management, can only be performed efficiently and rapidly if the processor hardware includes capabilities to support the OS. Virtually all processors include such capabilities to a greater or lesser extent, including virtual memory management hardware and process management hardware. The hardware includes special purpose registers and buffers, as well as circuitry to perform basic resource management tasks.
- One of the most important functions of the OS is the scheduling of processes, or tasks. The OS determines which process should run at any given time. Typically, the hardware will interrupt a running process from time to time to enable the OS to make a new scheduling decision so as to share processor time fairly among a number of processes.
- Another important OS function is memory management. Most contemporary operating systems include a virtual memory capability, which has two benefits: (1) A process can run in main memory without all of the instructions and data for that program being present in main memory at one time, and (2) the total memory space available to a program may far exceed the actual main memory on the system. Although memory management is performed in software, the OS relies on hardware support in the processor, including paging and segmentation hardware.

Although the focus of this text is computer hardware, there is one area of software that needs to be addressed: the computer's operating system. The operating system is a program that manages the computer's resources, provides services for programmers, and schedules the execution of other programs. Some understanding of operating systems is essential to appreciate the mechanisms by which the CPU controls the computer system. In particular, explanations of the effect of interrupts and of the management of the memory hierarchy are best explained in this context.

The chapter begins with an overview and brief history of operating systems. The bulk of the chapter looks at the two operating system functions that are most relevant to the study of computer organization and architecture: scheduling and memory management.

8.1 OPERATING SYSTEM OVERVIEW

Operating System Objectives and Functions

An operating system is a program that controls the execution of application programs and acts as an interface between the user of a computer and the computer hardware. It can be thought of as having two objectives:

- **Convenience:** An operating system makes a computer more convenient to use.
- **Efficiency:** An operating system allows the computer system resources to be used in an efficient manner.

Let us examine these two aspects of an operating system in turn.

The Operating System as a User/Computer Interface The hardware and software used in providing applications to a user can be viewed in a layered or hierarchical fashion, as depicted in Figure 8.1. The user of those applications, the end user, generally is not concerned with the computer's architecture. Thus the end user views a computer system in terms of an application. That application can be expressed in a programming language and is developed by an application programmer. To develop an application program as a set of processor instructions that is completely responsible for controlling the computer hardware would be an overwhelmingly complex task. To ease this task, a set of systems programs is provided. Some of these programs are referred to as **utilities**. These implement frequently used functions that assist in program creation, the management of files, and the control of I/O devices. A programmer makes use of these facilities in developing an application, and the application, while it is running, invokes the utilities to perform certain functions. The most important system program is the operating system. The operating system masks the details of the hardware from the programmer and provides the programmer with a convenient interface for using the system. It acts as mediator, making it easier for the programmer and for application programs to access and use those facilities and services.

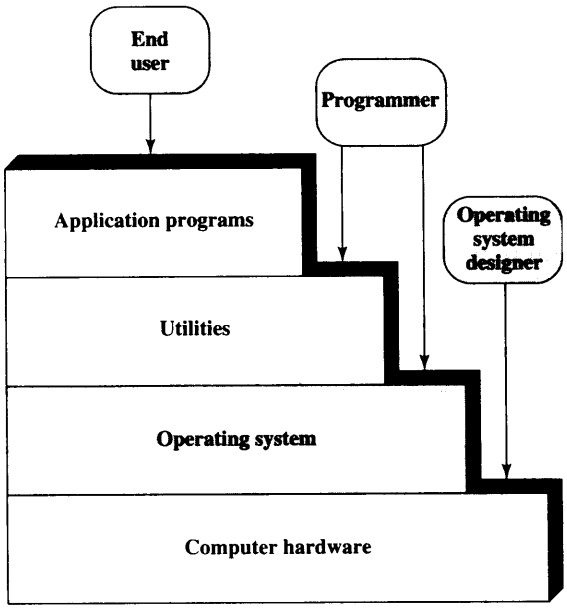


Figure 8.1 Layers and Views of a Computer System

Briefly, the operating system typically provides services in the following areas:

- **Program creation:** The operating system provides a variety of facilities and services, such as editors and debuggers, to assist the programmer in creating programs. Typically, these services are in the form of utility programs that are not actually part of the operating system but are accessible through the operating system.
- **Program execution:** A number of tasks need to be performed to execute a program. Instructions and data must be loaded into main memory, I/O devices and files must be initialized, and other resources must be prepared. The operating system handles all of this for the user.
- **Access to I/O devices:** Each I/O device requires its own specific set of instructions or control signals for operation. The operating system takes care of the details so that the programmer can think in terms of simple reads and writes.
- **Controlled access to files:** In the case of files, control must include an understanding of not only the nature of the I/O device (disk drive, tape drive) but also the file format on the storage medium. Again, the operating system worries about the details. Further, in the case of a system with multiple simultaneous users, the operating system can provide protection mechanisms to control access to the files.
- **System access:** In the case of a shared or public system, the operating system controls access to the system as a whole and to specific system resources. The access function must provide protection of resources and data from unauthorized users and must resolve conflicts for resource contention.
- **Error detection and response:** A variety of errors can occur while a computer system is running. These include internal and external hardware errors, such as a memory error, or a device failure or malfunction; and various software errors, such as arithmetic overflow, attempt to access forbidden memory location, and inability of the operating system to grant the request of an application. In each case, the operating system must make the response that clears the error condition with the least impact on running applications. The response may range from ending the program that caused the error, to retrying the operation, to simply reporting the error to the application.
- **Accounting:** A good operating system collects usage statistics for various resources and monitor performance parameters such as response time. On any system, this information is useful in anticipating the need for future enhancements and in tuning the system to improve performance. On a multiuser system, the information can be used for billing purposes.

The Operating System as Resource Manager A computer is a set of resources for the movement, storage, and processing of data and for the control of these functions. The operating system is responsible for managing these resources.

Can we say that it is the operating system that controls the movement, storage, and processing of data? From one point of view, the answer is yes: By managing the

computer's resources, the operating system is in control of the computer's basic functions. But this control is exercised in a curious way. Normally, we think of a control mechanism as something external to that which is controlled, or at least as something that is a distinct and separate part of that which is controlled. (For example, a residential heating system is controlled by a thermostat, which is completely distinct from the heat-generation and heat-distribution apparatus.) This is not the case with the operating system, which as a control mechanism is unusual in two respects:

- The operating system functions in the same way as ordinary computer software; that is, it is a program executed by the processor.
- The operating system frequently relinquishes control and must depend on the processor to allow it to regain control.

The operating system is, in fact, nothing more than a computer program. Like other computer programs, it provides instructions for the processor. The key difference is in the intent of the program. The operating system directs the processor in the use of the other system resources and in the timing of its execution of other programs. But in order for the processor to do any of these things, it must cease executing the operating system program and execute other programs. Thus, the operating system relinquishes control for the processor to do some "useful" work and then resumes control long enough to prepare the processor to do the next piece of work. The mechanisms involved in all this should become clear as the chapter proceeds.

Figure 8.2 suggests the main resources that are managed by the operating system. A portion of the operating system is in main memory. This includes the **kernel**, or

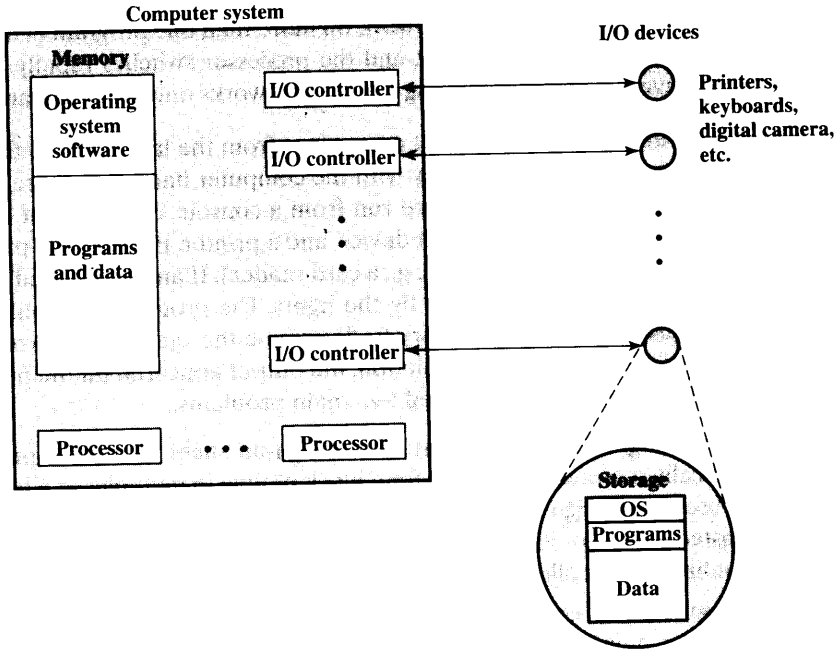


Figure 8.2 The Operating System as Resource Manager

nucleus, which contains the most frequently used functions in the operating system and, at a given time, other portions of the operating system currently in use. The remainder of main memory contains user programs and data. The allocation of this resource (main memory) is controlled jointly by the operating system and memory-management hardware in the processor, as we shall see. The operating system decides when an I/O device can be used by a program in execution, and controls access to and use of files. The processor itself is a resource, and the operating system must determine how much processor time is to be devoted to the execution of a particular user program. In the case of a multiple-processor system, this decision must span all of the processors.

Types of Operating Systems

Certain key characteristics serve to differentiate various types of operating systems. The characteristics fall along two independent dimensions. The first dimension specifies whether the system is batch or interactive. In an **interactive** system, the user/programmer interacts directly with the computer, usually through a keyboard/display terminal, to request the execution of a job or to perform a transaction. Furthermore, the user may, depending on the nature of the application, communicate with the computer during the execution of the job. A **batch** system is the opposite of interactive. The user's program is batched together with programs from other users and submitted by a computer operator. After the program is completed, results are printed out for the user. Pure batch systems are rare today. However, it will be useful to the description of contemporary operating systems to examine batch systems briefly.

An independent dimension specifies whether the system employs **multiprogramming** or not. With multiprogramming, the attempt is made to keep the processor as busy as possible, by having it work on more than one program at a time. Several programs are loaded into memory, and the processor switches rapidly among them. The alternative is a **uniprogramming** system that works only one program at a time.

Early Systems With the earliest computers, from the late 1940s to the mid-1950s, the programmer interacted directly with the computer hardware; there was no operating system. These processors were run from a console, consisting of display lights, toggle switches, some form of input device, and a printer. Programs in processor code were loaded via the input device (e.g., a card reader). If an error halted the program, the error condition was indicated by the lights. The programmer could proceed to examine registers and main memory to determine the cause of the error. If the program proceeded to a normal completion, the output appeared on the printer.

These early systems presented two main problems.:

- **Scheduling:** Most installations used a sign-up sheet to reserve processor time. Typically, a user could sign up for a block of time in multiples of a half hour or so. A user might sign up for an hour and finish in 45 minutes; this would result in wasted computer idle time. On the other hand, the user might run into problems, not finish in the allotted time, and be forced to stop before resolving the problem.
- **Setup time:** A single program, called a **job**, could involve loading the compiler plus the high-level language program (source program) into memory, saving the compiled program (object program), and then loading and linking together

the object program and common functions. Each of these steps could involve mounting or dismounting tapes, or setting up card decks. If an error occurred, the hapless user typically had to go back to the beginning of the setup sequence. Thus a considerable amount of time was spent just in setting up the program to run.

This mode of operation could be termed serial processing, reflecting the fact that users have access to the computer in series. Over time, various system software tools were developed to attempt to make serial processing more efficient. These include libraries of common functions, linkers, loaders, debuggers, and I/O driver routines that were available as common software for all users.

Simple Batch Systems Early processors were very expensive, and therefore it was important to maximize processor utilization. The wasted time due to scheduling and setup time was unacceptable.

To improve utilization, simple batch operating systems were developed. With such a system, also called a **monitor**, the user no longer has direct access to the processor. Rather, the user submits the job on cards or tape to a computer operator, who *batches* the jobs together sequentially and places the entire batch on an input device, for use by the monitor.

To understand how this scheme works, let us look at it from two points of view: that of the monitor and that of the processor. From the point of view of the monitor, it is the monitor that controls the sequence of events. For this to be so, much of the monitor must always be in main memory and available for execution (Figure 8.3). That portion is referred to as the **resident monitor**. The rest of the monitor consists of utilities and common functions that are loaded as subroutines to the user program at the beginning of any job that requires them. The monitor reads in jobs one at a time from the input device (typically a card reader

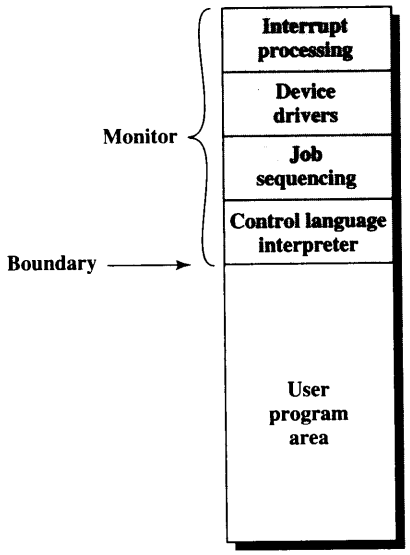


Figure 8.3 Memory Layout for a Resident Monitor

or magnetic tape drive). As it is read in, the current job is placed in the user program area, and control is passed to this job. When the job is completed, it returns control to the monitor, which immediately reads in the next job. The results of each job are printed out for delivery to the user.

Now consider this sequence from the point of view of the processor. At a certain point in time, the processor is executing instructions from the portion of main memory containing the monitor. These instructions cause the next job to be read in to another portion of main memory. Once a job has been read in, the processor will encounter in the monitor a branch instruction that instructs the processor to continue execution at the start of the user program. The processor will then execute the instruction in the user's program until it encounters an ending or error condition. Either event causes the processor to fetch its next instruction from the monitor program. Thus the phrase "control is passed to a job" simply means that the processor is now fetching and executing instructions in a user program, and "control is returned to the monitor" means that the processor is now fetching and executing instructions from the monitor program.

It should be clear that the monitor handles the scheduling problem. A batch of jobs is queued up, and jobs are executed as rapidly as possible, with no intervening idle time.

How about the job setup time? The monitor handles this as well. With each job, instructions are included in a **job control language** (JCL). This is a special type of programming language used to provide instructions to the monitor. A simple example is that of a user submitting a program written in FORTRAN plus some data to be used by the program. Each FORTRAN instruction and each item of data is on a separate punched card or a separate record on tape. In addition to FORTRAN and data lines, the job includes job control instructions, which are denoted by the beginning "$". The overall format of the job looks like this:

```
$JOB
$FTN
⋮          } FORTRAN instructions
$LOAD
$RUN
⋮          } Data
$END
```

To execute this job, the monitor reads the $FTN line and loads the appropriate compiler from its mass storage (usually tape). The compiler translates the user's program into object code, which is stored in memory or mass storage. If it is stored in memory, the operation is referred to as "compile, load, and go." If it is stored on tape, then the $LOAD instruction is required. This instruction is read by the monitor, which regains control after the compile operation. The monitor invokes the loader, which loads the object program into memory in place of the compiler and transfers control to it. In this manner, a large segment of main memory can be shared among different subsystems, although only one such subsystem could be resident and executing at a time.

We see that the monitor, or batch operating system, is simply a computer program. It relies on the ability of the processor to fetch instructions from